SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **43**:153–185
Published online 17 February 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2107

JavAdaptor—Flexible runtime updates of Java applications

Mario Pukall[1,*,†], Christian Kästner[2], Walter Cazzola[3],
Sebastian Götz[4], Alexander Grebhahn[1], Reimar Schröter[1] and Gunter Saake[1]

[1]*School of Computer Science, University of Magdeburg, Magdeburg, Sachsen-Anhalt, Germany*
[2]*Philipps-University Marburg, Marburg, Germany*
[3]*University of Milano, Milano, Italy*
[4]*University of Dresden, Dresden, Germany*

SUMMARY

Software is changed frequently during its life cycle. New requirements come, and bugs must be fixed. To update an application, it usually must be stopped, patched, and restarted. This causes time periods of unavailability, which is always a problem for highly available applications. Even for the development of complex applications, restarts to test new program parts can be time consuming and annoying. Thus, we aim at dynamic software updates to update programs at runtime. There is a large body of research on dynamic software updates, but so far, existing approaches have shortcomings either in terms of flexibility or performance. In addition, some of them depend on specific runtime environments and dictate the program's architecture. We present JavAdaptor, the first runtime update approach based on Java that (a) offers flexible dynamic software updates, (b) is platform independent, (c) introduces only minimal performance overhead, and (d) does not dictate the program architecture. JavAdaptor combines schema changing class replacements by class renaming and caller updates with Java HotSwap using containers and proxies. It runs on top of all major standard Java virtual machines. We evaluate our approach's applicability and performance in non-trivial case studies and compare it with existing dynamic software update approaches.

Received 4 November 2010; Revised 13 December 2011; Accepted 19 December 2011

KEY WORDS: dynamic software updates; program evolution; state migration: tool support

1. INTRODUCTION

Once a program goes live and works in productive mode, its development is not completed. It has to be changed because of bugs and new requirements. To maintain a program, it usually must be stopped, patched, and restarted. This downtime is always a problem for applications that must be highly available. But, also for the development of complex applications, restarts to test the new program parts can be time consuming and annoying. This is also true for end-user desktop applications that have to be restarted because patches must be applied [1]; end users prefer update approaches that do not interrupt their tasks. For these reasons, we aim at *dynamic software updates* (DSU), that is, program updates at runtime.

Even though dynamic languages such as SMALLTALK, PYTHON, or RUBY natively support runtime program changes, we address Java for several reasons. First, Java is a programming language commonly used to implement highly available applications. Examples are *Apache Tomcat*, *Java DB*, or *JBoss Application Server*. Second, in most fields of application, Java programs execute faster than programs based on dynamic languages [2]. Thus, developers often prefer Java over

*Correspondence to: Mario Pukall, School of Computer Science, University of Magdeburg, P.O. Box 4120, Magdeburg, Sachsen-Anhalt, Germany.

†E-mail: mario.pukall@iti.cs.uni-magdeburg.de

dynamic languages in time-critical scenarios. Amongst others, one reason for the better performance is that Java is a statically typed language. Unfortunately, compilation prevents Java and other statically typed languages such as C or C++ from natively offering powerful instruments for runtime program updates.

Literature suggests a wide range of DSU approaches for Java (see related work in Section 6). The *flexibility* of an approach can be determined by answering the following three questions: Are unanticipated changes allowed (i.e., can we apply requirements for which the running program was not prepared)? Can already loaded classes (including their schema) be changed, and is the program state kept beyond the update? Other quality criteria for a DSU approach are the caused *performance* overhead, the influence on the *program architecture*, and the *platform independency*. We believe that it is impossible to prepare an application for all potential upcoming requirements. Furthermore, only offering modifications of not previously executed program parts while disregarding the executed parts (e.g., already loaded classes) restricts the application of program changes. In addition, state loss and major performance overhead are unacceptable in many scenarios as well. Next, we argue that DSU approaches should not dictate the program's architecture, that is, they should be capable of being integrated into the program's natural architecture (different application domains might require different architectures). Last but not least, runtime update approaches should not force the customer to use a specific platform for program execution, for example, to use a Linux-based Java virtual machine even though the customer only runs Windows-based machines. For all these reasons, we aim at (a) flexible, (b) platform independent, and (c) performant runtime update approaches that (d) do not affect the program's natural architecture. However, we do not (yet) aim at a solution that fully supports reliable (immediate) and consistent runtime updates (which, to our best knowledge, is not supported by any existing DSU approach, which is applicable in real-world scenarios). In other words, *our goal is to provide Java with the same runtime update capabilities known from dynamic languages*.

Researchers spent much time to overcome Java's shortcomings regarding runtime program adaptation. Approaches such as *Javassist* [3, 4] and *BCEL* [5] allow to apply some unanticipated changes, but only to program parts that have not been executed yet. In contrast, *Steamloom* [6], *Reflex* [7], *PROSE* [8], *DUSC* [9], *AspectWerkz* [10], *Wool* [11], or *JAsCo* [12] allow unanticipated changes even of executed program parts; however, Steamloom, Reflex, PROSE, AspectWerkz, Wool, and JAsCo do not support class schema changing runtime updates. Although DUSC allows class schema changes, the program loses its state. Another dynamic software update approach is *JRebel* [13], which puts abstraction layers between the executed code and the JVM. It enables class schema changes except from modifications of the inheritance hierarchy. Kim presents in [14], a DSU approach based on proxies, which, similar to JRebel, only enables schema changes that do not affect the inheritance hierarchy.

We present JAVADAPTOR, the first (to our best knowledge) dynamic software update approach that fulfills all our quality criteria postulated earlier: it is flexible, platform independent, performant, and it does not affect the architecture of the program to be updated. To meet the criteria, we utilize Java HotSwap in an innovative way and combine it with class replacement mechanisms. Technically, we update all classes with a changed schema via class replacements and update their callers with the help of Java HotSwap. The key concepts of our solution are class renamings (to replace classes) and containers, respectively, proxies (to avoid caller class replacements). Furthermore, we contribute a discussion of desired properties for DSU approaches and a detailed survey of related approaches and their trade offs. In addition, we demonstrate the practicability of our approach with non-trivial case studies and show that the performance drops are minimal. We, last but not least, discuss ongoing and future work to improve JAVADAPTOR.

2. MOTIVATING EXAMPLE

Program maintenance is not a trivial task, which usually affects many parts of a program. Depending on the requirements, it ranges from single statement modifications to complex structural modifications, that is, it might affect all language constructs of Java as listed in Table I.

DOI: 10.1002/spe

Table I. Language constructs of Java [15].

Construct to be changed	Related elements
Classes	
(1) Class declaration	Modifiers, generic, inner classes, superclass, subclasses, superinterfaces, class body, member declarations
(2) Class members	Fields, methods
(3) Field declarations	Modifiers, field initialization, field type
(4) Method declarations	Modifiers, signature (name, parameters), return type, throws, method body
(5) Constructor declarations	Modifiers, signature (name, parameter), throws, constructor body
(6) Blocks	Statements
Interfaces	
(7) Enums	Enum declaration, enum body
(8) Interface declaration	Modifiers, generic, superinterface, subinterface, interface body, member declarations
(9) Interface members	Fields, method declarations
(10) Field (Constant) declarations	Field initialization, field type
(11) Abstract method declarations	Signature (name, parameters), return type, throws
(12) Blocks	Statements
(13) Annotations	Annotation type, annotation element

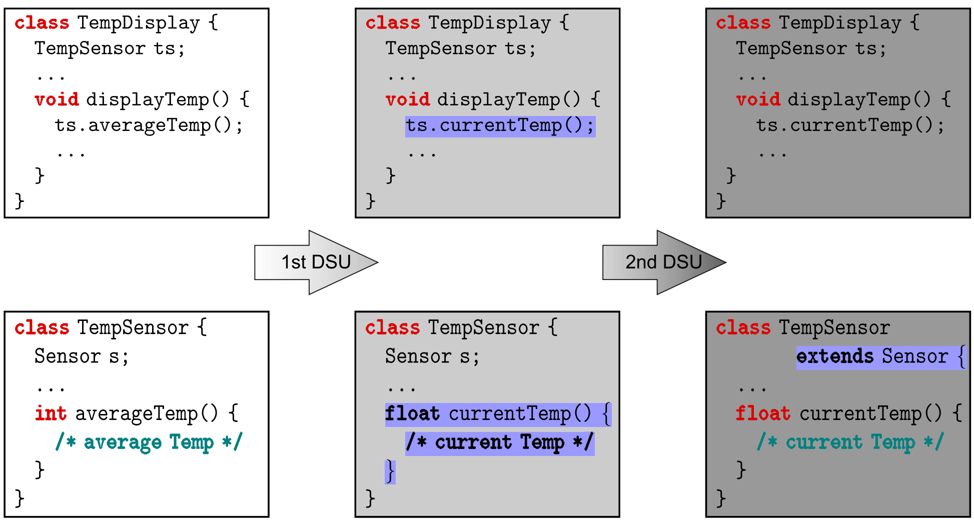

Figure 1. Weather station. The example spans updates, which replace methods, remove fields, and change inheritance hierarchies.

The weather station program depicted in Figure 1 exemplifies that even simple program changes can affect many parts of a program. The weather station program consists of two classes. One class (`TempSensor`) measures the air temperature whereas the other class (`TempDisplay`) is responsible for displaying the temperature. Consider a maintenance task: the actual measuring algorithm (average temperature) must be replaced by another measuring algorithm (current temperature). Because the service provided by the weather station must be non-stop available, stopping the program to apply the necessary changes is no option; we want to change it at runtime. The application of the new functionality requires to change different parts of the program. First, method `averageTemp` of class `TempSensor` must be replaced by method `currentTemp`, which requires to change the class schema. Second, to execute the new algorithm, method `displayTemp` of class `TempDisplay` must be reimplemented. Short time after applying the new measuring algorithm, it was also decided to let `TempSensor` inherit from class `Sensor` to add new functions to `TempSensor` while avoiding to implement them again. Therefore, statement `extends`

`Sensor` has to be applied to class `TempSensor`. Additionally, member `s` of original class `TempSensor` has to be removed because superclass `Sensor` let it become useless. However, changing the program code is only the first step toward an updated application. In addition, all objects that exist in the program must be also updated to let them access the new program parts as well as to keep the program state.

Even if the required program changes seem to be simple, they affect many different parts of the program (i.e., points 1–6 of Table I). Therefore, we search for a new mechanism in Java that allows to change every part of a program at runtime without anticipating the changes.

3. THE JAVA VIRTUAL MACHINE

To understand what is provided or possible in Java and what challenges remain regarding runtime adaptation, it is necessary to understand the standard design of Java's runtime environment—the Java virtual machine (JVM) [16]. As shown in Figure 2, a Java program is stored in the *heap*, in the *method area*, as well as on the *stacks* of the JVM. Within the heap, the runtime data of all class instances are stored [17]. In case a new class instance has to be created, the JVM explicitly allocates heap memory for the instance, whereas the *garbage collector* cleans the heap from data bound to class instances no longer used by the program. Unlike the heap, the method area stores all class (type) specific data such as runtime constant pool, static field information and method data, and the code of methods (including constructors) [17]. The stacks contain the currently executed program parts.

Changing a program during its execution in the JVM requires to modify the data within the heap, the method area, and on the stacks. For instance, program changes such as those depicted in Figure 1, which also include method replacements require to extensively change the data of a class. In general, they require to modify the class schema. Unfortunately, the JVM does not permit class schema changes, because class schema changes may let the data on the stack, on the heap, and the class data stored in the method area become inconsistent whereas the JVM does not provide functions to synchronize them. To disallow the developer class schema changing updates, the JVM enforces a strict class loading concept. To load a class, the JVM requests the following basic class loaders (in this order): (a) the *bootstrap* class loader (root class loader—loads system classes), (b) the *extension* class loader (loads classes of the extension library), and (c) the *application* class loader (loads classes from classpath). The first class loader in this hierarchy that is able to load the requested

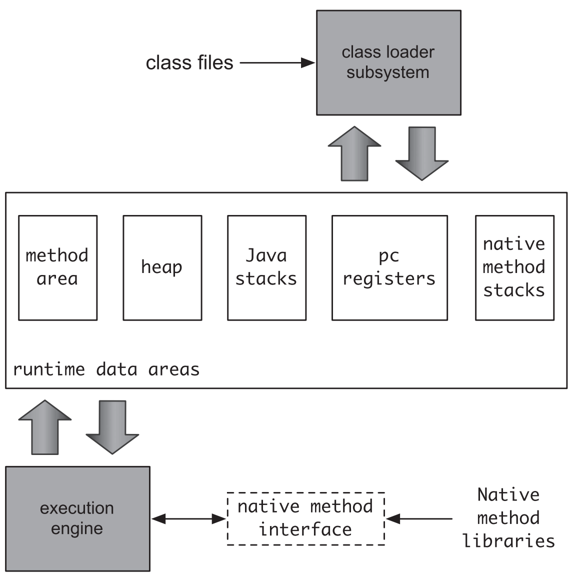

Figure 2. Program representation—HotSpot JVM [16].

DOI: 10.1002/spe

class will be finally bounded to this class, that is, none of the other class loaders is allowed to load or reload this class. The only way (besides customized class loaders that we will discuss in later sections) to reload (update) a class with a changed schema is to unload the old class version, which is only possible if the owning class loader can be garbage collected. Unfortunately, a class loader can only be garbage collected if all classes (even the unchanged ones) loaded by this class loader are dereferenced, which is equivalent to a (partial) application stop.

Java HotSwap. Despite the insufficient native runtime adaptation support of the JVM, there is one feature that provides some simple runtime update capabilities—called *Java HotSwap* [18]. It is provided by the Java virtual machine tool interface (JVMTI) [19] and allows to replace the body of a method (which partly covers points 4–6 of Table I) while the program is running. Even if HotSwap is not a standard feature, it is implemented by all major Java virtual machines commonly used in production, that is, the HotSpot JVM, the JRockit JVM, and IBM's JVM.

The class data restructuring via Java HotSwap consists of the following steps: First, an updated version of a class is loaded into the JVM. It contains the new method bodies. Second, it is checked if old and new class versions share the same class schema. Third, the references to the constant pool, method array, and method objects of the old class are successively (in the given order) redirected to their (up-to-date) counterparts within the updated class. After this is done, all corresponding method calls refer to the redefined methods. Unfortunately, Java HotSwap (and other features of JVMTI) neither allows to swap the complete class data nor remove or add methods, that is, it does not allow class schema changes.

4. DYNAMIC SOFTWARE UPDATES VIA JAVADAPTOR

Having described the shortcomings of Java's runtime environment, that is, the JVM, regarding flexible runtime program updates, we present JAVADAPTOR, which overcomes the limitations of the Java VM and adds flexible DSU to Java while not causing platform dependencies, architecture dependencies, and significant performance drops. It combines Java HotSwap and class replacements, which are implemented via containers and proxies.

4.1. Tool description and demonstration

Before we describe the concepts of our DSU approach, let us illustrate the general architecture and update process of JAVADAPTOR.

Tool description. Figure 3 describes JAVADAPTOR from the developers point of view. The current implementation of our tool comes as a plug-in, which smoothly integrates into the *Eclipse IDE*[‡] (conceptually, JAVADAPTOR could be integrated into any other IDE or even used without an IDE). The implementation of the required program updates conforms to the usual static software development process, that is, the developer implements the required functions using the Eclipse IDE and compiles the sources. This ensures type safety because of the static type checking done by the compiler.

When the developer decides to update the running application, JAVADAPTOR establishes a connection to the JVM executing the application (see Figure 4). In more detail, it connects to the JVM's JVMTI, which is used to control the JVM [19] (accessible from outside the JVM through the *Java debug interface*, which is part of the *Java platform debugger architecture* [20]). Once the update process is triggered, JAVADAPTOR prepares the classes changed within Eclipse so that they can be applied to the running application. The required bytecode modifications are performed by Javassist.[§] To load and instantiate new class versions, a special update thread is added to the target application. This thread is only active when the running program is updated and thus causes no performance penalties during normal program execution. After the update, JAVADAPTOR disconnects from the application. The described process can be repeated as often as required.

[‡]http://www.eclipse.org/
[§]http://www.csg.is.titech.ac.jp/chiba/javassist/

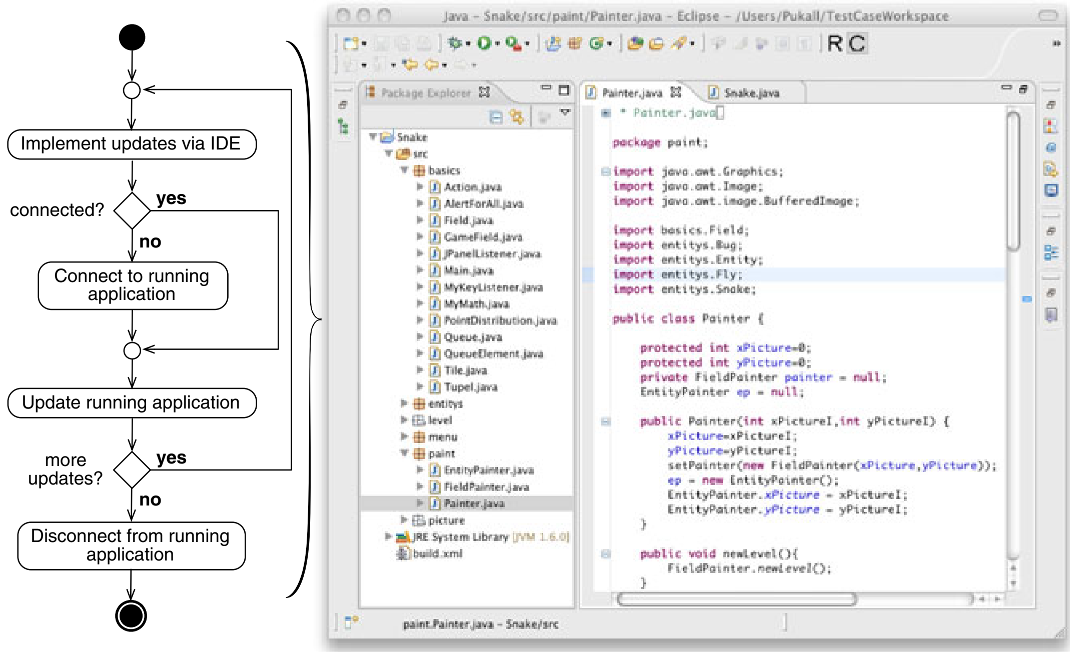

Figure 3. Update process.

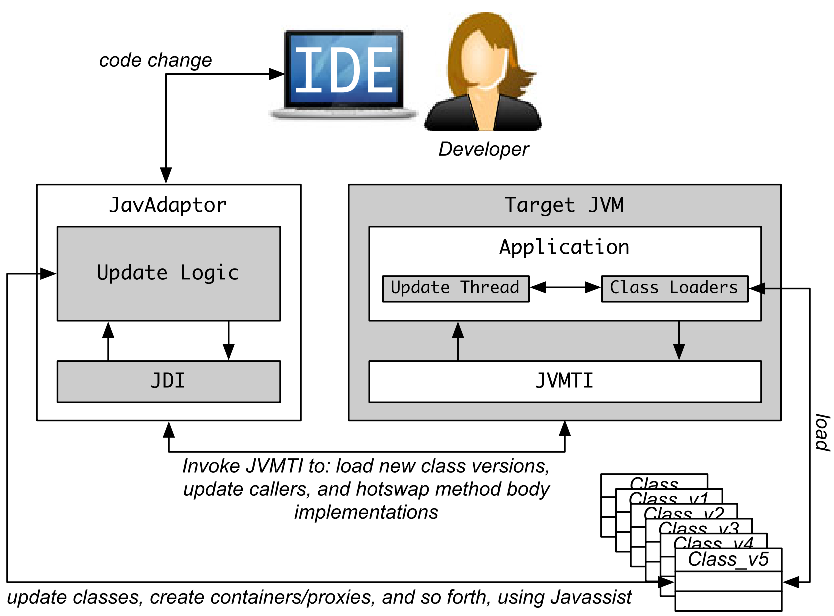

Figure 4. Dynamic software update architecture.

Tool demonstration. Because abstract descriptions on the usage of tools are sometimes hard to understand and do not reflect the reality well, we created a tool demonstration showing JAVADAPTOR in action. Concretely, we used JAVADAPTOR to update the well-known arcade game *Snake* at runtime. The update consists of four different steps, each adding new functions to the (at startup) very basic game. It required to redefine existing methods, to add new methods and fields, and even to update inheritance hierarchies. That is, the demonstration covers all kinds of

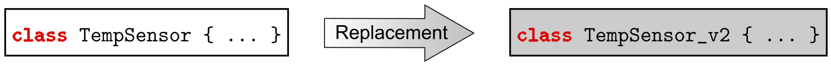

Figure 5. Class renaming.

updates essential to flexibly update running applications. For more information about our tool demo, see [21]. The corresponding demo video is available on YouTube.[¶]

In the following, we describe the basic mechanisms on how JAVADAPTOR changes applications running in the target JVM, namely class replacements using containers and proxies.

4.2. *Class reloading*

As stated in Section 3, the JVM disallows updating an already loaded class when the update alters the class schema. To circumvent these restrictions, we perform class replacements (updates) through *class renaming*. As exemplified in Figure 5, the key idea is that, although we cannot load a new class version with the same name, we rename the new version and load it under a fresh name. Because the resulting class name is not registered in any class loader, the updated class can be loaded by the same class loader that also loaded the original class.

Listing 1. JAVADAPTOR – class reloading.

```
1  class ClassUpdateLoader {
2   VirtualMachine targetJVM;
3   ...
4   void replaceClass(String oldClassName) {
5     if(isOldClassLoaded) {
6       ...
7       CtClass c = classpool.getCtClass(oldClassName);
8       c.replaceClassName (oldClassName, newClassName);
9       ...
10      ReferenceType refT = targetJVM.classesByName("UpdateHelper").get(0);
11      ObjectReference uHelper = refT.instances(0).get(0);
12      uHelper.invokeMethod(thread, loadClass, args[newClassName], options);
13    }
14  }
15 }
```

Listing 2. Target VM – class reloading.

```
16 class UpdateHelper extends Thread{
17  ClassLoader origClassLoader;
18  ...
19  void loadClass(String newClassName) {
20    origClassLoader.loadClass(newClassName);
21  }
22 }
```

Listing 1 sketches how class loading based on class renaming is implemented in JAVADAPTOR. The renamed and updated class (class `TempSensor_v2` from our motivating example depicted in Figure 1) is created by our adaptation tool (using the source level API of Javassist to manipulate the bytecode in Lines 7 and 8, Listing 1). In the next step, the adaptation tool invokes method `loadClass` (Line 12) of class `UpdateHelper` (Lines 16–22, Listing 1), which resides in the update thread added to the target application on application start. By invoking `loadClass` within the target application, the new class version is loaded by the same class loader that loaded the original class (Listing 2, Line 20), which ensures that our DSU approach is compatible with any application employing multiple class loaders (e.g., component-based applications).

[¶]http://www.youtube.com/watch?v=jZm0hvlhC-E

DOI: 10.1002/spe

4.3. Caller-side updates

As demonstrated earlier, our class reloading mechanism allows us to load a new version of an already loaded class even if the class schema has changed. However, the mechanism only triggers the loading of the updated class. To let the class become part of program execution, all references to the original class have to be changed to point to the new class version. For the sake of clarity, we will name the classes, which hold references to classes to be reloaded (updated) *callers* and the classes subject to updates *callees*. In addition, the terms *caller side* and *callee side* cover the class itself as well as all its instances.

In case of short-lived objects, such as local variable `local` of class `TempDisplay` (Figure 6), only method body redefinitions are required to refer to the new class version. This is, because with each method execution, the local variables are newly created. Thus, after redefining a method, such as depicted in Figure 6, the local variables created during method execution will be of the type of the updated class (`TempSensor_v2`). Those updates can be easily located and applied using the source level API of Javassist and Java HotSwap.

A snippet of the corresponding update code is depicted in Listing 3. For each application class, JAVADAPTOR checks whether the class references the class to be updated. Technically, all classes referenced by the caller side are requested using Javassist method `getRefClasses` (Line 4). If references of the old callee class (`TempSensor`) are found, we update them method by method to the updated class (Lines 17–24). After this is done, the updated caller method is redefined using Java HotSwap (Line 26).

Different from references to short-lived objects, references to long-lived objects (such as class or instance field references) are vital beyond method executions, that is, they are inherent parts of the caller side. Thus, caller-side updates because of references to long-lived objects of type of the callee must be handled in a different way. Those updates require four steps: (1) *caller detection*, (2) *instantiation of the updated callee class*, (3) *callee-side state mapping*, and (4) *reference updates*.

4.3.1. Caller detection. To replace the references to instances of the original callee class by instances of the new callee class version (as required for class `TempSensor` from our motivating example), we have to detect all callers and their instances that refer to long-lived objects of the original callee class. The JVMTI supports this operation. A snippet of the caller detection implementation is depicted in Listing 4. First, the class object of the old callee class is retrieved from the target JVM (Line 4). This object is used to get all instances of the old callee class via reflection (Line 5). Again, using the instances all callers are retrieved (Line 7). This includes even callers whose global fields are of type of a super class the old class extends, which is possible because the function requests the objects runtime type and not the static type. In addition, JAVADAPTOR searches all application classes for class and instance fields of type of the old callee class (using Javassist method `getRefClasses`). This is necessary to detect even caller classes which are not

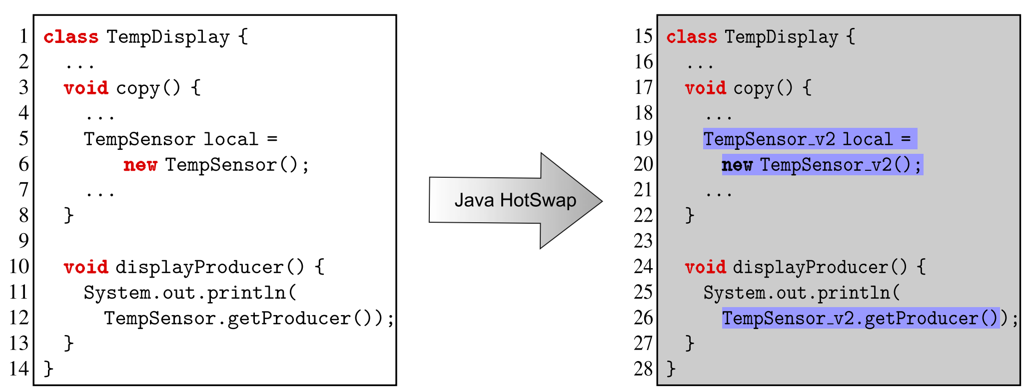

Figure 6. Caller side updates in case of short-lived objects.

Listing 3. JAVADAPTOR – caller-side update in case of short-lived objects.

```
class CallerUpdateShortLived {
 ...
 void detectAndUpdateCaller(CtClass caller) {
  Collection col = caller.getRefClasses();
  Iterator colIterator = col.iterator();
  ...
  while(colIterator.hasNext()) {
   CtClass callee = (CtClass) colIterator.getNext();
   if(callee.getName().compareTo(oldClassName) == 0) {
    ClassMap classMap = new ClassMap();
    classMap.put("oldClassName", "newClassName");
    ...
    callee.replaceClassName(classMap);
   }
  }
  ...
  CtMethod[] methods = caller.getDeclaredMethods();
  ...
  ExprEditor exprE = new ExprEditor() {...};
  for(int i = 0; i < methods.length; i++) {
   ...
   methods[i].instrument(exprE);
   ...
  }
  ...
  targetVM.redefineClasses(callerClass);
  ...
 }
}
```

Listing 4. JAVADAPTOR – caller detection.

```
class CallerUpdateLongLived {
 ...
 List<ClassObjectReference> detectCallers(String oldClassName) {
  ReferenceType refL = targetJVM.classesByName(oldClassName);
  List<ObjectReference> oRefL = refL.instances(0);
  ...
  List<ObjectReference> cRefL = oRefL.get(i).referringObjects(0);
  ...

 }
}
```

yet loaded, instantiated, or whose instances do not refer to the callee side because the corresponding class or instance fields are not yet initialized.

4.3.2. Callee class instantiation. In the next update step, JAVADAPTOR creates for each instance of the original callee class an instance of the new class version (`TempSensor_v2` from our motivation). The new instances will be used later on to replace the instances of the old class and thus to update the caller side (i.e., class `TempDisplay`).

Again, the instantiation is triggered by our adaptation tool. The corresponding code is depicted in Listing 5. Method `createInstance` of our update tool takes as argument the name of the new class version and invokes method `newInst` of class `UpdateHelper` in the target application, which creates an instance of the new class. Listing 6 shows a code snippet of method `newInst` of the helper class at application side. Via method `forName`, we retrieve the class object of the updated class (Line 14). Then, we call method `allocateInstance` of class `sun.misc.Unsafe`, which performs the instantiation. One reason why we use `sun.misc.Unsafe` instead of method `newInstance` of class `Class` for instantiation is that it prevents us from initializing the objects twice, that is, it would require to initialize the objects when they are created and again when they get the state from their outdated counterparts, which would be inefficient. Furthermore, method *allocateInstance* eases the instantiation of classes that do not provide a default constructor.

DOI: 10.1002/spe

Listing 5. JAVADAPTOR – instantiation.

```
1 class UpdateInstantiation {
2  ClassObjectReference updateHelper;
3  ...
4  void createInstance(String newClassName) {
5   ...
6   updateHelper.invokeMethod(thread, newInst, args[newClassName], options);
7  }
8 }
```

Listing 6. Target VM – instantiation.

```
9  class UpdateHelper extends Thread {
10  Unsafe unsafe;
11  ClassLoader applClassLoader;
12  ...
13  Object newInst(String cName) {
14   Class c = Class.forName(cName, false, applClassLoader);
15   return unsafe.allocateInstance(c);
16  }
17 }
```

4.3.3. Callee-side state mapping. Having finished the instantiation step, JAVADAPTOR has to map the state from old to corresponding new instances. In our example, this means to map the state from instances of old class `TempSensor` to instances of class update `TempSensor_v2`. Because of the simplicity of one-to-one mappings (mappings of values from fields that exist in both class versions) and mappings where either fields are removed or added, they can be executed automatically. However, for more complex (indefinite) mappings, for example, mappings where the type of a field differs between old and new class, but the field name remains the same, a mapping function must be manually defined by the user.

4.3.4. Reference updates. Finally, once for each instance of the original callee class an instance of the new class version has been created and initialized with the state of its outdated counterpart, JAVADAPTOR updates the caller side. That is, all instances of the original callee class (according our motivation class `TempSensor`) have to be replaced by the instances of the new callee class (class `TempSensor_v2`). Unfortunately, updated and outdated callee class are not type compatible, thus, objects of the updated class cannot be assigned to fields of the type of the outdated class (such as required to update field `ts` of caller class `TempDisplay`).

Containers. To solve the type-incompatibility problem while avoiding to change the caller class schema, we use containers whose usage is exemplified in Figure 7. Before program start, JAVADAPTOR prepares the program for the container approach, that is, it adds field `cont` (Line 17) to each class in the program. The container field does not affect program execution as long as no callee of the caller class has to be replaced. To replace a callee instance referenced by the caller class, the program has to be changed as depicted in the right part of Figure 7. First, JAVADAPTOR creates a container class (see Figure 7, Lines 48–52) used to store instances of the new callee class (`TempSensor_v2`). Second, our tool assigns the up-to-date counterpart of an outdated object (such as referenced by field `ts` in Figure 7) to an instance of the container. The container instance containing the up-to-date object is then assigned to field `cont` within the caller class (class `TempDisplay`). Third, the tool redirects all accesses of the old callee instance to the updated callee instance located in the container (see Figure 7, Lines 36–38 and Lines 43–44), that is, the tool redefines all method bodies in which the old callee instance is accessed and swaps the resulting method bodies via Java HotSwap. Note that we, for clarity reasons, will remove the necessary downcasts to the specific container type (as shown in Lines 37 and 43 of Figure 7) from the following code examples.

Proxies. The basic container approach described in Figure 7 is sufficient in many cases. However, it fails when the caller class to be updated contains methods whose parameters or returned objects are of the type of the old callee class (such as shown in Figure 8, Line 5 and 9). One workaround would

DOI: 10.1002/spe

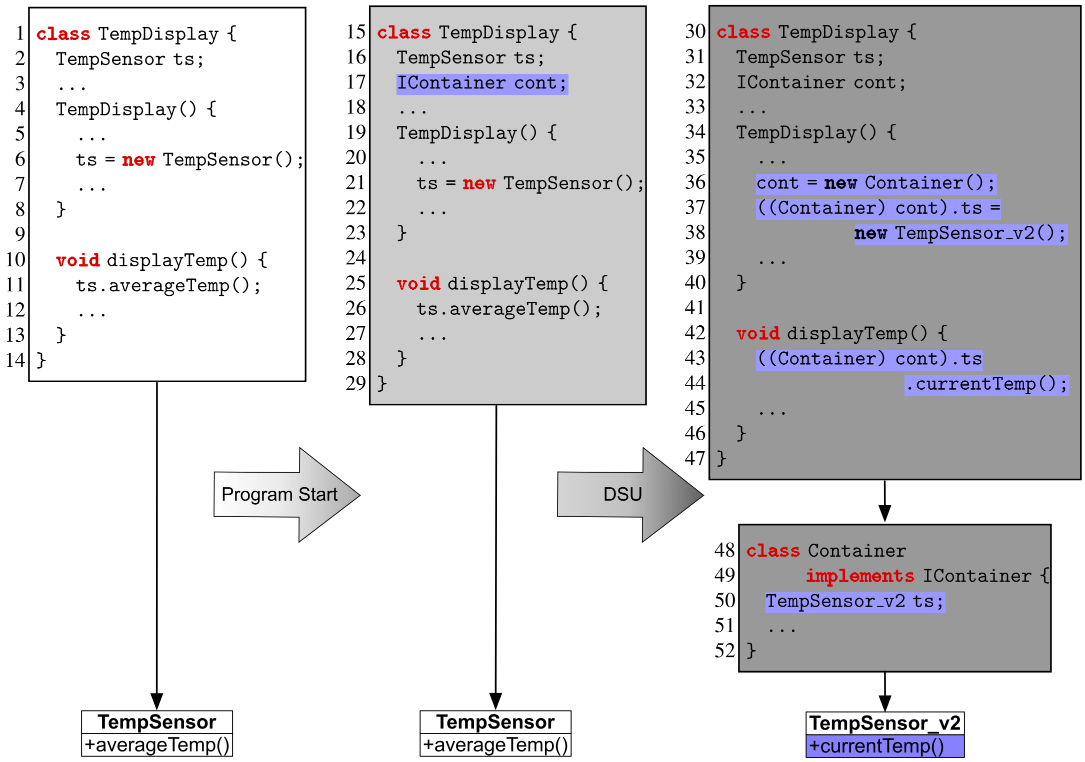

Figure 7. Caller-side updates through containers.

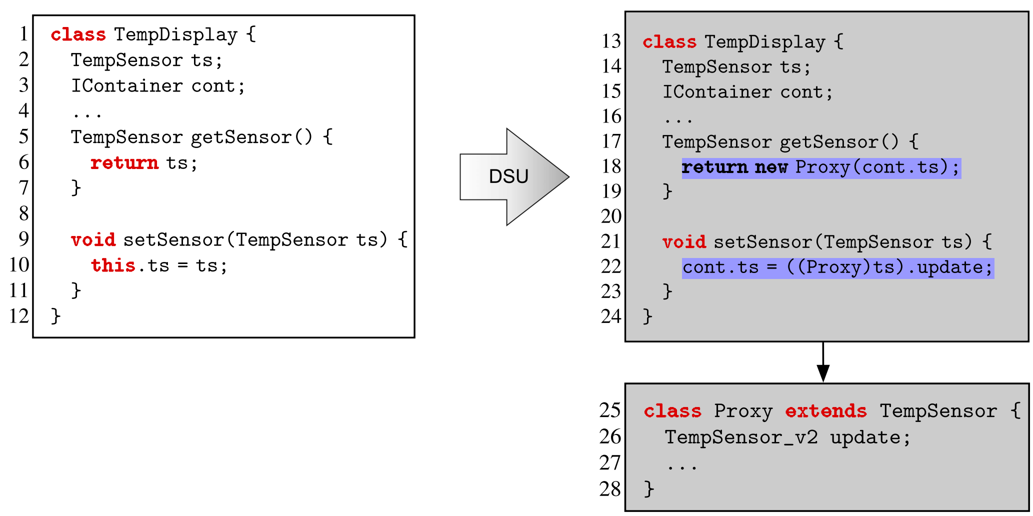

Figure 8. Caller-side updates using proxies.

be to replace the caller class as well. But, this strategy may result in additional class replacements, which at the worst require to essentially replace all classes of the system and thus let our DSU approach become inefficient. To avoid cascading class replacements, we extend our approach by proxies (see Figure 8). Caller updates work in the same manner as described earlier. Only difference is that, in addition to the container class, a proxy class is generated.

The idea of proxies is to guide objects of an updated callee class through the caller methods that require or return objects of the type of the old callee class. The usage of proxies is exemplified on the basis of method `getSensor` of class `TempDisplay`, which returns an instance of callee class

DOI: 10.1002/spe

TempSensor (Line 6). After replacing callee class TempSensor by class TempSensor_v2, method getSensor has to return an instance of the new callee class, which is not possible because TempSensor and TempSensor_v2 are not type compatible. To achieve type compatibility, we wrap the instance of TempSensor_v2 with an instance of class Proxy (Line 18). Because the proxy extends class TempSensor, it can be returned by method getSensor. Technically, we use method *allocateInstance* from class sun.misc.Unsafe for the proxy instantiation because it eases the creation of proxy instances even if the proxy's super class has no default constructor. To use the returned object wrapped by the proxy at receiver side (i.e., within the class that called method getSensor), the object is unwrapped. That is, the proxy is only used to guide instances of the new callee class through type incompatible methods. The receiver will finally work with the new callee object and not with the proxy object, which avoids the *self-problem* [22]. How to propagate instances of the updated callee class back to the caller (more precisely to the container) is exemplarily shown in Figure 8 (Line 22). Before method setSensor is called, its parameter (i.e, an instance of TempSensor_v2) is wrapped by a proxy. To unwrap and use the received instance of class TempSensor_v2, proxy ts must be cast to type Proxy.

Listing 7. Bytecode modifications proxy: return.

```
1  TempSensor getSensor() {
2     0 aload_0
3     1 astore_1
4     2 aconst_null
5     3 astore_2
6     4 aload_1
7     5 getfield #15 <TempDisplay.fieldContainer1265725244704>
8     8 checkcast #17 <TempDisplay_Cont_1>
9    11 getfield #21 <TempDisplay_Cont_1.ts>
10   14 astore_2
11   15 aload_2
12   16 invokestatic #27 <TempSensor_Proxy_1.newInst>
13   19 checkcast #29 <TempSensor>
14   22 areturn
15 }
```

Proxy bytecode modifications. Up to this point, most of the required bytecode modifications described earlier could be processed using the source level API of Javassist, which makes bytecode modifications easy to handle. However, the modifications required to apply proxies exceed the power of Javassist's source level API. The source level API cannot terminate the type of local variables referenced through the method's *local variable table*. Because parameters are stored in local variables by default, it is not possible to apply the code to unwrap them using the source level API. The same problem occurs when locally stored objects that have to be returned must be wrapped by a proxy. For that reasons, we manage the application of proxies manually, that is, with the bytecode level API of Javassist.

Listing 7 shows the bytecode modifications (getSensor of example class TempDisplay) required to wrap returned objects (Lines 12 and 13). First, we call method newInst (Line 12) of the Proxy class, which takes as parameter an object of the updated callee class (TempSensor_v2), wraps the object by a newly created proxy instance, and returns the proxy. Second, the returned proxy is casted to the type of the old callee class (TempSensor, Line 13).

How to modify the bytecode to unwrap proxy-based parameters (setSensor of example class TempDisplay) is depicted in Listing 8 (Lines 2–5). First, we load the parameter stored in a local variable (Line 2). Second, we cast the parameter to the related proxy type (Line 3). Third, we unwrap the updated class instance (TempSensor_v2) stored in field call of the proxy object (Line 4). Fourth, to avoid recurring unwrappings, the unwrapped instance is stored in the local variable that previously stored the proxy (Line 5).

4.3.5. Concurrent updates of multiple classes. So far, we described the mechanisms and concepts of JAVADAPTOR on the basis of the very simple weather station example given in Section 2. This

DOI: 10.1002/spe

Listing 8. Bytecode modifications proxy: parameters.

```
void setSensor(TempSensor ts) {
   0 aload_1
   1 checkcast #23 <TempSensor_Proxy_1>
   4 getfield #34 <TempSensor_Proxy_1.call>
   7 astore_1
   8 aload_0
   9 aload_1
  10 astore_3
  11 astore_2
  12 aload_2
  13 getfield #36 <TempDisplay.fieldContainer1265725244704>
  16 checkcast #17 <TempDisplay_Cont_1>
  19 aload_3
  20 putfield #38 <TempDisplay_Cont_1.ts>
  23 return
}
```

example only consists of one single class update and the corresponding caller-side update. However, JavAdaptor does not only allow the developer to update a single class, but multiple classes in one step, which is essential to update complex real-world applications. On the one hand, this is because updates of real-world applications normally span many different classes. On the other hand, concurrent updates of multiple classes is essential for inheritance hierarchy updates, because superclass updates implicitly require to update and reload corresponding subclasses, too (note that we have to reload the subclasses to let them `extend` the new superclass version).

Figure 9 sketches how JavAdaptor handles concurrent updates of multiple classes. At first, JavAdaptor reloads all classes with changed schemas (as described in Section 4.2). Afterwards, it identifies all classes (callers) with references to the classes to be reloaded (see Section 4.3.1). This information is gained in one atomic step for efficiency reasons. That is, having an overview about all changes required to update the running program allows us to create possible containers and proxies in one single step. In addition, we only have to touch each class one-time to modify its bytecode. However, in the next two steps, JavAdaptor creates the new callee instances and maps the state (as we described in Sections 4.3.2 and 4.3.3). If this is carried out, JavAdaptor updates all references conforming to the workflow described in Section 4.3.4. Because we already gained information about all dependencies between callers and callees, this can be efficiently carried out in one atomic step, too. In the last update step, we update all modified and hotswapable classes at once using Java HotSwap. This includes not only all callers of reloaded classes, but also classes that are explicitly changed by the developer.

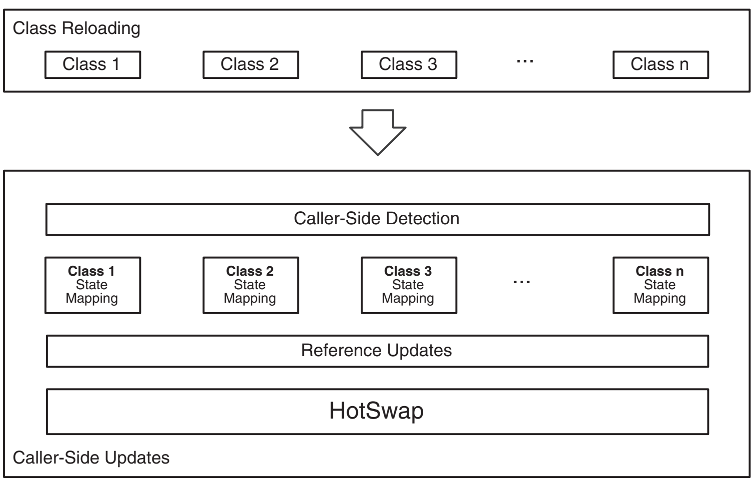

Figure 9. Concurrent multiple class updates.

DOI: 10.1002/spe

In summary, JAVADAPTOR allows us to flexibly change applications during their runtime. The update granularity can vary from minor changes (i.e., of single classes) to system wide changes (i.e., of multiple classes). In addition, JAVADAPTOR will only update the changed classes and the corresponding caller classes. All other classes remain untouched, which minimizes the influence of the update on the running program.

5. EVALUATION

Our goal was to develop an update approach which allows running Java applications to be updated in every possible way (a feature only known from dynamic languages). In addition, the approach should not introduce performance drops. To check whether JAVADAPTOR meets the goals, we applied it to different non-trivial case studies.

5.1. HyperSQL

To simulate a real-world scenario, which requires flexible runtime updates, we proceeded as follows. We chose a reasonable application to update, which was *HyperSQL*[‖] amongst others used by *OpenOffice* (we chose HyperSQL because it is a database management system for which runtime adaptation promises benefits of no-downtime, it is entirely written in Java, and an open source application whose source code is available for the latest program version and earlier versions). We started version 1.8.0.9 of it downloaded from the HyperSQL website and applied all changes to evolve it to the next version 1.8.0.10 without shutting down the application. After the program starts, we ran the open source database benchmark *PolePosition*[**] to generate and query some data, which ensured that HyperSQL was fully activated and deployed.

The new version of HyperSQL (released 9 months after version 1.8.0.9 came out) comes with a bunch of changes. It fixes major bugs that cause null-pointer exceptions, problems with views, timing issues, corrupted data files, and deadlocks. Additionally, new and improved functionality such as new lock-file implementations and performance improvements to the web server are included. To lift the running program from version 1.8.0.9 to the new version 1.8.0.10, we had to update 33 of 353 classes. The updates affected many language constructs (Points 1–7 of Table I). In case of 21 out of 33 classes, the changes did not affect the class schema, that is, the changes could be applied by our tool solely using Java HotSwap. Apart from that, 12 classes were affected by schema-changing program modifications. JAVADAPTOR replaced them using class replacements. The state mappings that came along with the replacements span one-to-one mappings, added, and removed fields, that is, they were automated by JAVADAPTOR. Table II lists all classes that had to be replaced. Note that updating class `NIOLockFile` also included changes to the inheritance hierarchy. In addition, with class `LockFile$HeartbeatRunner`, we had to update even a nested class. Inheritance hierarchy updates as well as updates that involve nested classes are supported by JAVADAPTOR. However, Table II provides also information about the required caller updates, that is, how many caller classes are updated in the context of short-lived objects, containers, or proxies. The number of references within method bodies that have to be changed to update the caller classes is given as well (in brackets). In 148 out of 197 cases (75.1%), updates because of references to short-lived callee objects (via Java HotSwap) were required to update the callers. 21 caller classes (10.7%) had to be updated through containers. 28 caller class updates (14.2%) required proxies.

To verify that HyperSQL was still correctly working (in a consistent state) after the update, we reran the PolePosition benchmark. In the result, HyperSQL passed the benchmark without errors, that is, all database operations were correctly executed after the update. In a second test we checked whether the updates were applied and active. Therefore, we hooked the JVM profiler *VisualVM*[††] into the running application and checked what classes/methods were executed during

[‖]http://hsqldb.org/
[**]http://polepos.sourceforge.net/
[††]https://visualvm.dev.java.net/

DOI: 10.1002/spe

Table II. HyperSQL: required class reloadings because of schema changes.

Replaced class Kind of update	Caller updates Short-lived objects (# Ref.)	Container (# Ref.)	Proxy (# Ref.)
FontDialogSwing structural	8 (9)	0 (–)	0 (–)
HsqlDatabaseProperties functional	11 (98)	2 (25)	11 (23)
LockFile functional	1 (9)	10 (5)	11 (47)
LockFile$HeartbeatRunner functional	2 (2)	0 (–)	0 (–)
Logger structural	22 (93)	3 (93)	3 (4)
NIOLockFile changed inheritance hierarchy	0 (-)	0 (–)	0 (–)
ScriptReaderZipped functional	3 (3)	0 (–)	0 (–)
SimpleLog structural	9 (105)	3 (27)	0 (–)
Token structural	5 (671)	0 (–)	0 (–)
Trace structural	80 (1306)	0 (–)	0 (–)
Transfer structural	4 (6)	0 (–)	0 (–)
View functional	3 (37)	3 (13)	3 (16)

The table lists all classes to be reloaded. It furthermore provides information on the required caller updates, that is, how many referring classes are updated in the context of short-lived objects, containers, or proxies. The number of updated references is given as well (in brackets).

the PolePosition benchmark. We found out that 5 of the 12 replaced classes were active and central part of program execution during the PolePosition benchmark which confirms that they were updated correctly. The remaining 7 classes were correctly loaded into the JVM, but inactive during the benchmark. Thus, we could not verify their correct execution.

5.2. *Refactorings*

With the HyperSQL case study we, demonstrated the flexibility and practicability of JAVADAPTOR on the basis of a real-world application. However, we could continue indefinitely updating specific real-world applications, which demonstrate the capabilities of our tool and would end up each time with just another case study. The problem with case studies such as HyperSQL is that they present specific update scenarios, which may not cover all eventualities and thus do not allow us to draw conclusions on the general applicability of JAVADAPTOR.

To get a better understanding of JAVADAPTOR's general applicability, we followed a different path and checked if the tool would be able to dynamically apply to common program updates, that is, updates that frequently occur in practice and do not rely on certain application scenarios. But, what are common program updates and how could we unbiased test if JAVADAPTOR is able to apply them to running applications? We found *Refactorings* [23] to be appropriate for our analysis. Actually, Dig and Johnson [24] found out that:

Refactorings cause more than 80% of API changes that were not backwards-compatible.

Once we decided to prove the general applicability of JAVADAPTOR on the basis of refactorings, we had to reason about a test setup, which ensures the tests to be unbiased. Our tests based on the refactorings presented by Fowler [25], which is the standard reference regarding refactorings. To

DOI: 10.1002/spe

achieve an unbiased test setup, we simply took the example programs from Fowler and refactored them at runtime. JAVADAPTOR was able to process all refactorings including possible state mappings. Out of the 72 refactorings, 61 (ca. 84%) required class reloadings. Reference updates because of class reloadings required containers in 57 out of 61 cases (ca. 93%) and proxies in three out of 61 cases (ca. 5%). State mappings could be automatically processed in 41 out of 61 cases (ca. 67%), whereas mapping methods were required in 20 out of 61 cases (ca. 33%).

To sum up, the results of our refactoring case study show that JAVADAPTOR covers a large bandwidth of different update scenarios, and chances are high that the tool performs well when it must update concrete real-world applications.

5.3. Performance

Having demonstrated JAVADAPTOR's capabilities regarding flexible updates, it is time to take a look at the performance penalties induced by our tool, that is, the execution speed of the changed program parts.

Because we were primarily interested in getting to know how JAVADAPTOR affects the performance of a real-world application, we chose our HyperSQL case study and proceeded as follows. We ran the PolePosition benchmark (mentioned earlier) immediately after runtime updating HyperSQL to version 1.8.0.10 and compared the results with the benchmark results of HyperSQL version 1.8.0.10 not updated at runtime. We could not measure any statistically significant difference, that is, the benchmark results of the HyperSQL instance updated at runtime were as good as the results of the HyperSQL instance not updated at runtime. In other words, the runtime updates we performed did not affect the performance of HyperSQL in a measurable way.

However, even if we did not measure performance penalties because of our runtime update approach in a real-world scenario, we assumed that our approach does not come entirely without performance overhead in some borderline cases. To get evidence about this assumption, we additionally implemented a micro benchmark that is able to detect even minimal performance penalties. It measures the costs of crossing the version barrier from old program parts (i.e., callers) to the new ones (i.e., updated callees).

To get reliable results, we ran 10 samples of one million invocations of all major invocation types and for each calculated the average access time in nanoseconds. For containers and local updates, no statistically significant performance overhead was measurable (calculated through a one-way analysis of variance), that is, programs updated using containers and local updates perform as fast as the original program. One reason for the good results is the just-in-time compiler of the JVM that is able to optimize the code used to instrument the containers.

In Section 4.3.4, we described the need for proxies to avoid implicit caller replacements in case the callee appears to be an argument of a caller method, returned by a caller method, or both. To figure out possible execution speed penalties owing to our proxy approach, we again ran 10 samples of one million (get, set, and set&get)-method invocations and recorded the method access times. The boxplots of the recorded access times are shown in Figure 10. The average access times with *no update* (left part of Figure 10) range from 13,73 ns to 13,93 ns, with a median access time value of 0 ns and only 2,7% to 3% outliers. When we reload the *Callee* and thus have to use proxies, the average method access times increase (middle of Figure 10) now ranging from 38 ns to 53,5 ns, whereas the median is still at 0 ns (7,3%–9,4% outliers). That is, dynamic updates involving proxies introduce slight execution speed penalties.

To get to know how the results scale, we put some workload on the methods and let them process statement `System.out.println('Hello JAVADAPTOR')`. The results are shown in Figure 11. As one can see, the times to execute the method bodies are much higher than the method access times, which results in similar overall method execution times with and without proxies, ranging from 8892 ns to 11,210 ns on average. That is, workload on methods (which should be the common case) renders the performance overhead introduced by proxies negligible. In addition, reloading the referring class (i.e., the *Caller*) as well, almost recovers the original method access times (right part of Figure 10). The average access times after reloading the caller, range from 12,79 ns to 19,79 ns, with a median access time value of 0 ns (2,9%–4,2% outliers).

DOI: 10.1002/spe

	Method	Median in ns	Mean in ns	Outliers in %
No Update	get	0	13,8	2,7
	set	0	13,93	3
	set&get	0	13,73	2,9
Callee	get	0	46	9,4
	set	0	38	7,3
	set&get	0	53,5	9,1
Caller	get	0	12,79	2,9
	set	0	15,61	3,3
	set&get	0	19,79	4,2

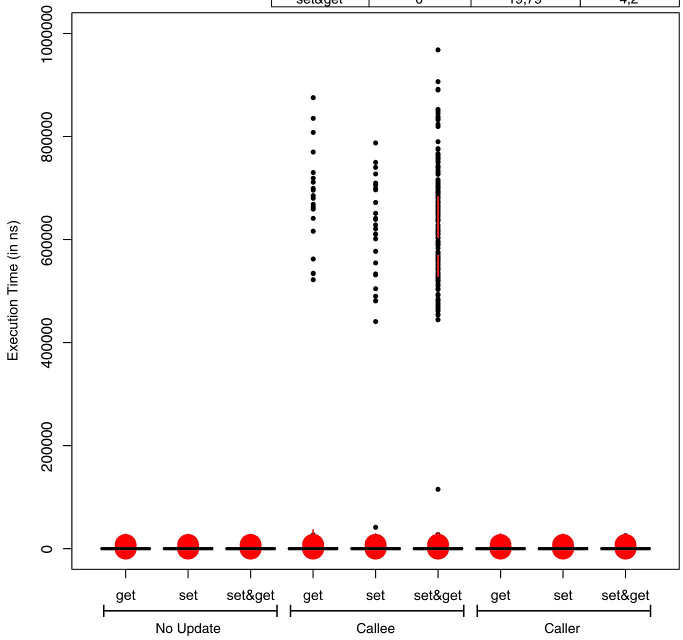

Figure 10. Method execution times in the presence of proxies. Meaning of the plotted elements: —— = box plot, ● = outlier, — = low concentration of overlapping outliers, ● = high concentration of overlapping outliers.

All in all, the results of our HyperSQL case study and the micro benchmark confirm that runtime program changes by JAVADAPTOR produce only minimal performance overhead. Only proxies produce a measurable overhead. Caller updates through local changes and containers do not cause measurable performance drops.

5.4. Update speed

Even if the contributions of our current JAVADAPTOR implementation are others than applying updates the fastest way, we evaluated how well JAVADAPTOR performs in this regard. That is, we measured the time JAVADAPTOR pauses the application during the update process to avoid program inconsistencies. Our measurements were based on two different programs representing different application scenarios.

At first, we measured the time required to update our HyperSQL case study under different conditions. With our first test, we measured the time period required to update HyperSQL with an empty database (i.e., without any data object stored), which was 1407 ms. In further tests, we ran the PolePosition benchmark creating hundreds, thousands, ten thousands, and hundred thousands of data objects before the update. The corresponding update times ranged from 1518 ms to 5346 ms, which seems to be not outstanding fast, but sufficient in many scenarios. By contrast, restarts and reinitializations of HyperSQL (e.g., filling caches, reloading data objects, creating views, creating users, etc.), as we simulated them using PolePosition, took more time.

The other application for which we measured the update times was the *Snake* demo we briefly described in Section 4.1 and presented in [21]. Compared with the update of HyperSQL, which affects wide parts of the system (the update spans changes made during 9 months of development),

DOI: 10.1002/spe

	Method	Median in ns	Mean in ns	Outliers in %
No Update	get	11196	11094	4,7
	set	10729	10805	0,9
	set&get	10729	10811	2,9
Callee	get	10730	11210	1,3
	set	10729	10922	3
	set&get	10729	10897	46
Caller	get	10729	8892	0,2
	set	10730	11134	5,5
	set&get	10730	11168	7,1

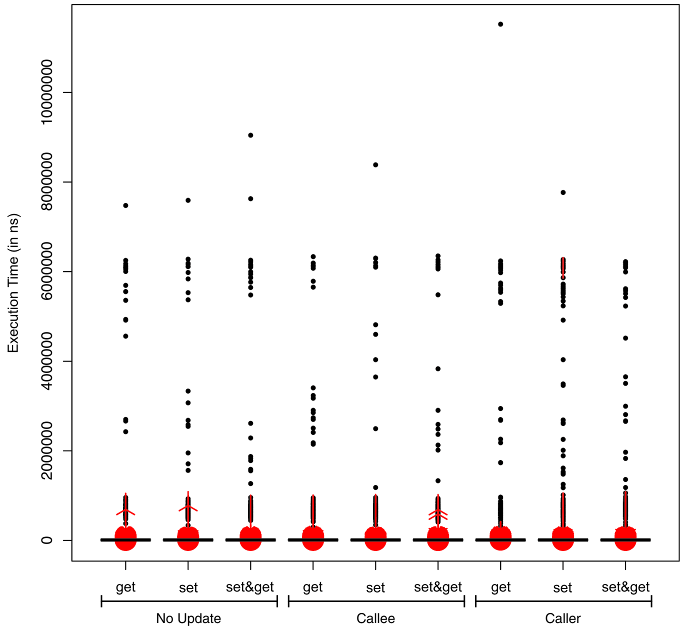

Figure 11. Method execution times in the presence of proxies and workload. Meaning of the plotted elements: —— = box plot, ● = outlier, — = low concentration of overlapping outliers, ● = high concentration of overlapping outliers.

each Snake update step consists only of small changes to few classes. Thus, the Snake updates represent scenarios common to the software development process, that is, frequent minor changes and immediate application of the changes. As our demo video (available on YouTube[‡‡]) suggests, the update times are rather short ranging from 28 ms to 142 ms.

All in all, the update times we measured suggest that our current JAVADAPTOR implementation could be beneficial in many different scenarios (even if currently other DSU approaches such as presented in [26] and [27] may offer shorter update times). The bottleneck of our current JAVADAPTOR implementation is JDI method `referringObjects`, which JAVADAPTOR uses during state mapping to identify the callers of an outdated object. The execution times of this method notably increase the more objects are present in the JVM, even if the number of objects to be updated remains unchanged. However, high speed updates were not yet in our scope. Therefore, our current JAVADAPTOR implementation is not optimized for them. But of course, optimizations to the update speed are subject to new versions of the JAVADAPTOR we are working on (we will discuss possible improvements of JAVADAPTOR and first benchmark results in Section 7).

6. RELATED WORK AND COMPARISON

In this section, we provide an overview of recent work to overcome Java's limitations regarding dynamic software updates. For better comparability and because of the broad range of related work ranging from theoretical to practical solutions, we focus on practice-oriented approaches such as JAVADAPTOR, which can be directly applied in real-world scenarios. We group the related work into three groups based on their main strategies: *customized Java virtual machines*,

[‡‡]http://www.youtube.com/watch?v=jZm0hvlhC-E

DOI: 10.1002/spe

customized class loaders, and *wrappers*. For each group, we discuss the general mechanism and some representative approaches.

In addition, we evaluate the quality of JAVADAPTOR and of the related work based on the criteria given in Section 1. That is, we analyze an approach's *flexibility*, *platform dependency*, *performance* and its influence on the *program architecture*. We chose the criteria because they let us describe the differences between the approaches. For instance, considering support for reliable (immediate) consistent program updates would be irrelevant, because no approach fulfills this criterion. Furthermore, the criteria align with our goals presented in Section 1. We derived the criterion *flexibility* from the fact that static software development allows the developer to change a program in any way, no matter *when* and *where* the changes must be applied. Runtime update approaches should provide the same flexibility to cover all update scenarios. We further choose *platform dependency* because platform independence is one of the reasons for the success of Java, that is, DSU approaches should not cause dependencies to specific JVM implementations. In Section 1, we argued that Java's *performance* in terms of program execution speed is better than the performance of dynamic languages, which natively provide flexible runtime updates. Ending up with an updated Java program whose execution speed is worse than the execution speed of the same updated program based on a dynamic language might be a good reason to prefer dynamic languages. Users virtually always prefer a good performing approach over a comparable, but worse performing one (particularly when the program is supposed to be used in production). Finally, we pick up the *program architecture* criterion because in software development, there is no such thing as 'one architecture fits all scenarios'. As already mentioned in Section 1, different scenarios require different architectures. Thus, DSU approaches should not restrict the usage of different architectures. However, different criteria might be of different importance to different stakeholders. For instance, users might emphasize *flexibility* whereas administrators might attach great importance to *platform independence*. That is, to satisfy the stakeholders, a DSU approach must fulfill all mentioned criteria.

6.1. Customized Java virtual machines

As mentioned in Section 3, the JVM disallows the developer to reload a class whose schema has changed and thus forbids flexible dynamic software updates.

Therefore, researchers suggest virtual machine patches that enable to reload classes with changed schemas. For instance, Malabarba *et al*. [28] added dynamic class loaders to their *dynamic virtual machine* (DVM) for this purpose. *JDrums* [29] is a JVM that uses handles to decouple classes and objects from each other to reload classes. The *Jvolve* VM [30] decouples classes using meta-objects that can be easily changed to refer to updated classes. In addition to Java HotSwap, which allows the developer to redefine methods bodies of already loaded classes, Dmitriev [18] patched the Hotspot JVM in such way that it supports even class schema changes. Unfortunately, unlike Java HotSwap, this feature never made it into a standard JVM.

Flexibility. All in all, customized Java virtual machines perform well when it comes to flexibility. They allow unanticipated changes of virtually all parts of a program. Furthermore, they all provide mechanisms to keep the program state beyond the update. Customized JVMs provide this flexibility because the update mechanism is implemented within the JVM itself and not at application level which otherwise would complicate or prevent flexible updates.

Platform dependency. Even if virtual machine customization seems to be the most natural way to enhance Java's runtime update capabilities (because it does not require to operate at application level to apply the update approach), different problems arise from it. First of all, there is a standard, which precisely defines the functionality and structure of a JVM [17]. Changing the standard to add dynamic software updates is difficult because it would require to change all existing JVM implementations. Thus, there are only slight chances that DSU becomes a standard. However, as long as DSU is not part of the JVM specification, it must be added via patches. One problem with JVM patches is that they are based on a specific JVM implementation and might not be applicable to other JVMs. In addition, each new release of the JVM must be patched again. This might be difficult (eventually impossible) in case the JVM implementation has largely changed in the new JVM version. Last but not least, companies rather prefer standard (certified) JVMs over customized

DOI: 10.1002/spe

ones to run their applications in productive mode. This is why dynamic software update approaches are needed that operate on top of different standard virtual machines.

Performance. First of all, we point out that it is virtually impossible to exactly measure and compare the performance of the referred approaches. Some JVMs are not available for download, and those that are available do (partly) support only outdated Java versions (e.g., JDRUMS only executes programs based on Java version 1.2). Thus, we were not able to benchmark them and get meaningful benchmark results. Instead, we searched the literature for information regarding the performance. We found that the four patched JVMs significantly differ in terms of performance (see [30] and [26]). *DVM* [28] executes programs in interpreted mode only, which is commonly known to be slow. *JDrums* [29] aims at lazy updates and uses transformer functions to migrate the state from old objects to their updated counterparts, which introduces noticeable constant performance overhead. *Jvolve* [30] immediately updates applications, that is, it applies the updates in one step and thus avoids considerable performance penalties. Würthinger *et al*. presented in [26] a new and improved version of Dmitriev's JVM patch [18] that comes without any performance overhead.

Program architecture. As previously described, JVM customization aims at integrating the update mechanisms with the JVM which makes changes to the application architecture unnecessary.

6.2. *Customized class loaders*

As mentioned earlier, the basic idea of JVM patches is to enhance the JVM with capabilities to reload and thus update classes. In addition to the basic class loaders required to load and run a program, the class loading capabilities of a program can be extended even at application level by customized class loaders [31], which is common technique to load updated versions of already loaded classes or components. For instance, the *OSGi Service Platform* [32] or Oracles *FastSwap* [33] utilize customized class loaders to update components. *Javeleon* [27] allows to flexibly update *NetBeans*-based applications and thus uses customized class loaders, too. Zhang and Huang [34] presented *dynamic update transactions* (*DUT*), which also make use of customized class loaders.

Flexibility. Customized class loaders serve the flexibility required to largely update running programs, that is, they allow to update virtually all parts of a running program in an unanticipated way while preserving the program state. This is true for *Javeleon* [27] and also for DUT [34]. In case of the *OSGi service platform* [32], the state of a bundle is lost when it is refreshed, although.

Platform dependency. Because customization of class loaders is a standard feature in Java, it can be applied to all standard Java runtime environments. *Javeleon* additionally requires *NetBeans* components for execution.

Performance. One issue with customized class loaders is that they reduce the application performance when applied to JVMs older than version 1.6. This is because old and updated program parts are loaded by different class loaders, which requires poor performing reflection-based version-barrier crossings. Cazzola [35] found out that even simple reflective method invocations (as required for crossing the version barrier) slow down method invocations with a factor of up to 6.5 compared with direct method invocations. More complex version-barrier crossings might cause even higher performance penalties. However, with Java 1.6, this situation relaxed because the related JVM is able to optimize reflective calls.

Program architecture. Generally, the application of customized class loaders largely affects the application architecture. More precisely, customized class loaders dictate how an application must be designed and thus disallow alternative (tailor-made) designs. DUT requires methods that maintain the updates to be present in each class. *Javeleon*, *FastSwap* as well as the *OSGi service platform* require the applications to run on top of their infrastructure to be refactored into components (if not already done). This does not only alter the application architecture, it might be also inefficient because even small changes require to replace whole components.

6.3. *Wrappers*

Another frequently used approach to provide Java with enhanced runtime update capabilities are wrappers (also known as decorators [36]). Wrappers aim at wrapping old program parts to update them [37–39].

To apply the updates introduced by a wrapper, all clients (callers) of the changed program parts must be updated, too. That is, all references to the original callee must be redirected to the corresponding wrapper instance that wraps the callee. To update the caller side, Gamma *et al*. [36] suggest that wrapper and wrappee extend the same superclass or (even more flexible) implement the same interface. The application of the wrapping can be either statically predefined before program start or triggered at runtime using method body redefinitions based on Java HotSwap (as we did it in [37]). However, the big conceptual drawback compared with JAVADAPTOR, JVM patches, and customized class loaders is that wrappers never really update (reload) classes, but put them in a new context from which several limitations (particularly regarding our criteria) arise.

Flexibility. Wrappers do not provide the same flexibility as JAVADAPTOR, customized JVMs and customized class loaders do. *Lasagne* [39] and *JAC* [38] only allow anticipated runtime program updates because the wrappings must be predefined before application start. Nevertheless, wrappers can be also used in an unanticipated way, as we demonstrated in prior work[37]. The big issue is that conceptually, wrappers cannot remove fields or methods defined in classes they wrap.

Platform dependency. The wrapper approach is a well-known design pattern [36], which is fully implemented at application level and thus does not require specific platforms to act. However, to enlarge its flexibility, it must be combined with techniques, which allow to (re-)define wrappings at runtime.

Performance. There is one point with wrappers that cause significant performance penalties: indirections owing to object wrappings. In [40], we measured the performance penalties caused by long wrapping chains, which was raised by up to 50% compared with the same program without wrappers.

Program architecture. The principle drawback of wrappers is that an application must be completely refactored to prepare it for wrapper-based dynamic software updates. If the developer aims at avoiding poor performing reflective field accesses, she has to allow read and write access to all fields of the old program part, namely the object to be wrapped. Furthermore, all classes have to be forced to implement unique interfaces. In addition, all fields have to be of the type of the interface their classes implement. That is, similar to customized class loaders, the wrapper approach dictates the design of an application and thus, restricts user-defined application designs. In addition, the forced design has serious drawbacks because it violates encapsulation and causes the self-problem [22]. Another problem with the design of several wrapper approaches is decreased reliability caused by frequent type casts.

6.4. JAVADAPTOR

So far, all considered approaches have their strengths and weaknesses regarding the given criteria, that is, no approach fulfills them all. But, as we described earlier, there is a need for approaches that cover all criteria. In the following, we compare JAVADAPTOR with the previously described approaches and discuss whether JAVADAPTOR fulfills all criteria or not. An overview of the comparison results can be found in Table III.

Flexibility. As demonstrated in Section 5, the flexibility of our runtime update approach JAVADAPTOR is as good as the flexibility that could be achieved by patched JVMs and customized class loaders. More precisely, it is at par with *Jvolve*, *JDrums*, *DVM*, the HotSpot VM patch of Dmitriev and Würthinger [18, 26], *Javeleon*, and *DUT*.

Platform dependency. When it comes to platform independence, JAVADAPTOR clearly outperforms many competitors. Without any JVM patches, it runs on top of all standard JVMs that provide Java HotSwap, which amongst others is a standard feature in the HotSpot VM, the JRockit VM, and IBM's JVM. Furthermore, it does not require any library or framework to act.

Performance. Another strength of JAVADAPTOR is its performance. As our benchmark results in Section 5 show, container-based updates come along without performance penalties, and proxy-based updates only cause slight performance drops. JAVADAPTOR neither requires performance-dropping JVM patches nor reflection-based version-barrier crossings (which may be slow particularly on older JVMs) caused by customized class loaders. It also does not depend on a component framework, such as *Javeleon*, *FastSwap* or *OSGi*, whose execution causes additional

DOI: 10.1002/spe

Table III. Overview comparison.

	DSU approach	Flexibility	Platform independency	Performance	Architecture independency
	JAVADAPTOR	●	●	●	●
JVM	Jvolve	●	○	●	●
	HotSpot	●	○	●	●
	JDrums	●	○	○	●
	DVM	●	○	○	●
CCL	Javeleon	●	●	◑	○
	DUT	●	●	◑	○
	FastSwap	◑	●	◑	○
	OSGi	○	●	◑	○
Wrapper	[37]	◑	●	○	○
	JAC	○	●	○	○
	Lasagne	○	●	○	○

performance overhead. Furthermore, JAVADAPTOR causes no wrapping chains and thus comes without the related performance issues.

Program architecture. Unlike customized JVMs, JAVADAPTOR requires to add a container field to each class. However, the container field is transparent to the user and can be easily integrated with the application to be updated without any changes to the architecture. By contrast, customized class loaders particularly in conjunction with component frameworks dictate the application design and thus render alternative (tailor-made) application designs impossible. This is also true for wrappers where the forced application design additionally causes serious drawbacks (for further details see Section 6.3).

7. ENHANCEMENTS AND OPTIMIZATIONS

In Section 4, we described the basic concepts of our DSU approach and evaluated its practicability under real-world conditions in Section 5. Even if the results of our evaluation confirm the practicability and usefulness of JAVADAPTOR, there is still space for improvements. In this section, we summarize work in progress to improve JAVADAPTOR. We point out that most of the discussed improvements here are inspired by existing work such as presented in [14, 25, 27]. However, we do not simply discuss the related work, but describe how to combine it with the existing JAVADAPTOR concept.

7.1. Update-speed improvements

In Section 5.4, we evaluated the update speed of JAVADAPTOR on the basis of our HyperSQL and Snake case studies. We found the current JAVADAPTOR implementation acceptably fast in this regard, but stated that it could be further improved. From what we found out, the bottleneck of our current JAVADAPTOR implementation is JVMTI method `referringObjects`, which helps us to identify all objects referring to outdated objects. The problem with method `referringObjects` is that it performs a full heap search every time we request the referring objects of an outdated object, which causes long program update times and thus long time periods of program unavailability if the program heap is large and/or many requests must be processed.

An appropriate solution to the described problem are *lazy state mappings* as Kim [14] and Gregersen [27] use them in their DSU approaches. Different from our current implementation, in which we map the state and update the referring program parts in one atomic step, lazy state mappings operate on a per access basis. That is, the state transfer between the outdated and up-to-date object and the update of the referring program parts is carried out from within the program if and only if an outdated object is accessed.

DOI: 10.1002/spe

Figure 12 exemplifies how lazy state mappings work and how we are going to integrate them into JavAdaptor. To dynamically change our small weather station program such that it computes and displays *current* instead of the *average* temperatures, JavAdaptor updates the running program as follows. It processes all update steps we described in Section 4, but applies additional code to the program, which carries out the state mapping and updates the referring program parts without the need of method `referringObjects`. More precisely, JavAdaptor modifies the program code in such a way that before each access to a potentially outdated object, it will be checked, whether the object must be updated or not. In the example depicted in Figure 12, this applies to all references to field `ts`, which we must update using our container approach because we replaced class `TempSensor` with class version `TempSensor_v2` add new method `currentTemp`. Concretely, before we access the up-to-date object (`TempSensor_v2`) stored in the container, we check whether the container object already exists and is up-to-date or not (see Figure 12, Line 27). In the latter case, a mapping method (in our example method `mapState`) of the container class will be called (Figure 12, Line 28). This method maps the state from outdated object (`TempSensor`) to the up-to-date object (i.e., of type `TempSensor_v2`), applies the newly created object to a container instance, and returns the container instance (see Figure 12, Lines 36–44). After the state mapping, the newly created object can be accessed as usual, that is, via the container instance (see Line 30).

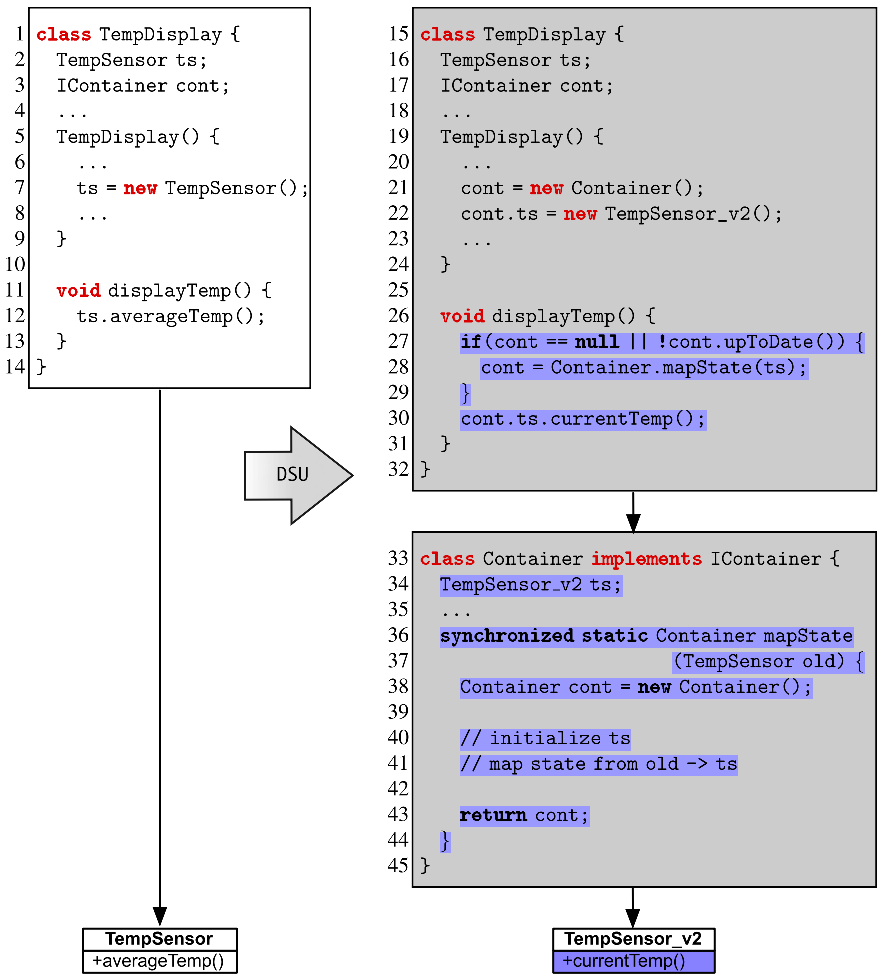

Figure 12. Lazy state mapping.

DOI: 10.1002/spe

After we described how we could provide our tool with lazy state mappings, let us present some update-speed numbers confirming that lazily mapping the state and thus avoiding to use method `referringObjects` could significantly improve the update speed of JAVADAPTOR. In Section 5.4, we measured the update times of JAVADAPTOR regarding our HyperSQL case study with zero, hundreds, thousands, ten thousands, and hundred thousands of data objects. The numbers ranged from 1407 to 5346 ms. With a JAVADAPTOR prototype that provides lazy state mappings as we sketched them in Figure 12, we were able to significantly reduce the update-speed times. Figure 13 contrasts the old update-speed times with the new ones based on lazy state mappings. What can be seen is, that the update-speed numbers remain somewhat comparable as long as only few objects are on the heap of the JVM. But, in case of many objects on the heap (here, hundred thousands of data objects), JAVADAPTOR based on lazy state mappings clearly outperforms our current (i.e., *busy*) state mapping implementation, that is,the prototype requires to pause the application only 916 ms, whereas current JAVADAPTOR causes an application pause time of 5346 ms.

The numbers presented in Figure 14 further underpin the benefit of lazy state mappings. Different from our HyperSQL case study, where the number of objects to be updated remained unchanged with each benchmark configuration, the results presented here outline how the application pause times develop depending on the number of objects scheduled for an update. As shown in Figure 14, the application pause times caused by our current JAVADAPTOR implementation further increase depending on the number of objects to be updated, which is because with each object update, JAVADAPTOR must call method `referringObjects`. By contrast, the application pause times of our JAVADAPTOR prototype based on lazy state mappings are significantly shorter and moreover, remain virtually unchanged regardless of the number of objects that require an update.

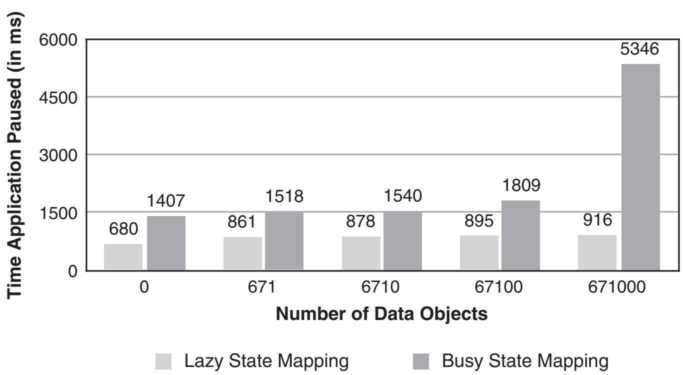

Figure 13. Update speed HyperSQL: lazy vs. busy state mapping.

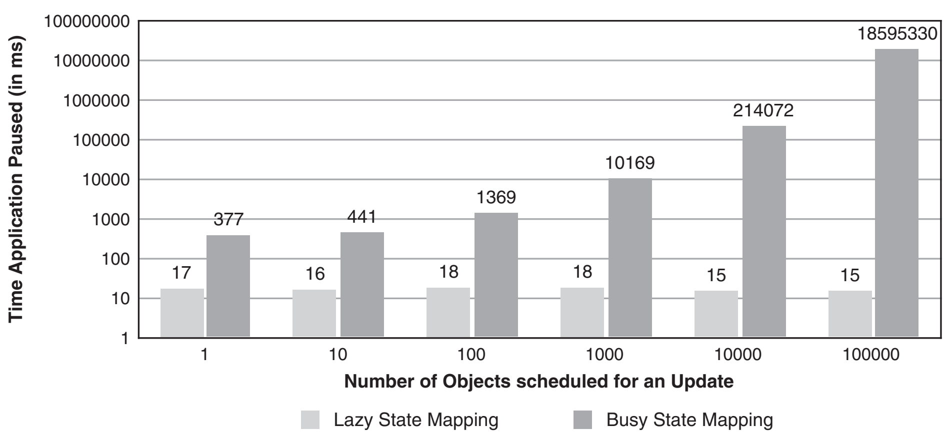

Figure 14. Update speed in correspondence to the number of objects to be updated: lazy versus busy state mapping.

Because the tests with our prototype show significant update-speed improvements, we are currently working to complete the integration of lazy state mappings into JAVADAPTOR. What is still missing is support for lazy state mappings within the Java system classes. Nevertheless, we are optimistic to provide a fully working JAVADAPTOR version with lazy state mappings soon.

7.2. Solutions towards consistent program updates

The HyperSQL as well as the Snake case study show that JAVADAPTOR could update programs without compromising their correctness, that is, the programs consistency. This is, because JAVADAPTOR already includes mechanisms aiming at consistent program updates. For instance, JAVADAPTOR permits updates only if the program sources compile without errors. Another example is, that JAVADAPTOR pauses the application during the update to ensure that all changed program parts are present within the JVM. However, similar to other DSU approaches, the current JAVADAPTOR implementation does not ensure program consistency at all beyond the update. Therefore, we discuss how to improve our tool in this regard.

7.2.1. Thread-safe updates. One issue we plan to tackle with future JAVADAPTOR versions is the lack of support for thread-safe updates of multi-threaded applications. Currently, updates of multithreaded applications may cause deadlocks and thus inconsistencies under certain conditions. Such a scenario is depicted in Figure 15. In the example, two different threads alternately access `TempSensor ts` of class `TempDisplay` of our small weather station. The first thread periodically instructs `ts` to measure the temperature (by calling method `measureTemp`), whereas the second thread is responsible for displaying the measured temperature (by calling method `displayTemp`). Because measuring and displaying the temperature at the same time would cause unexpected program behavior, access to `TempSensor ts` must be synchronized (see Figure 15, Lines 6–10 and Lines 14–17).

What could happen when JAVADAPTOR updates a multi-threaded application such as shown in Figure 15 (note that for clarity reasons, the lazy state mapping related code is hidden) is that, for some methods, the necessary method body redefinitions already took effect, whereas other methods

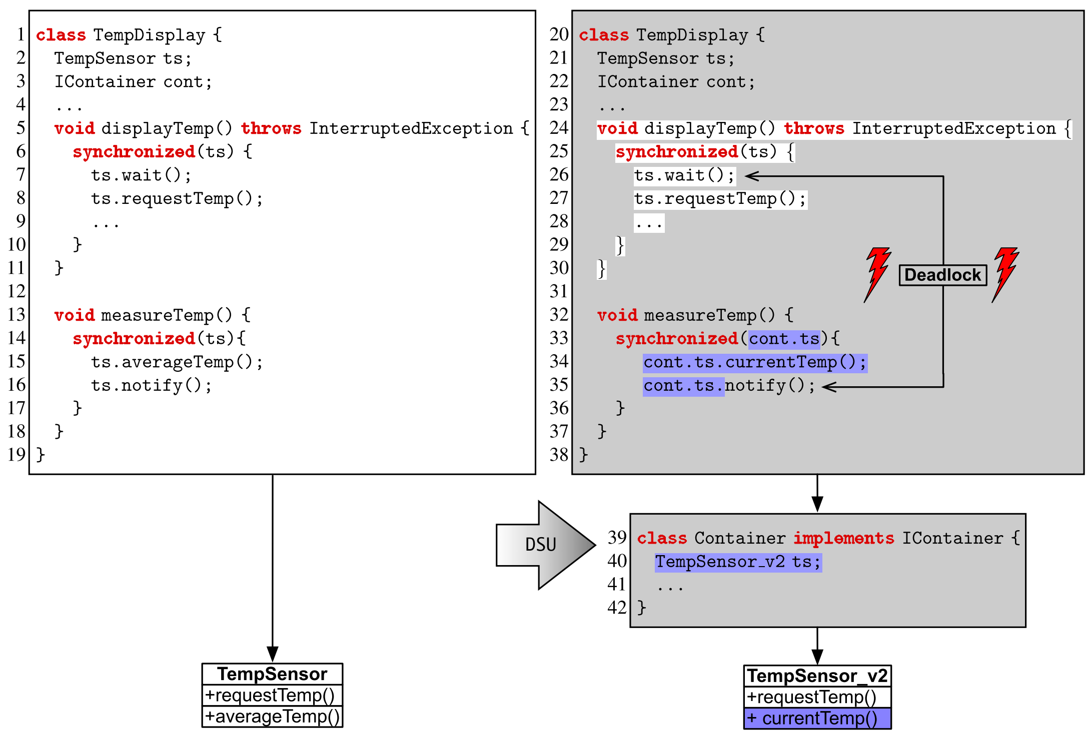

Figure 15. Deadlocks because of dynamic software updates.

DOI: 10.1002/spe

remain unaffected, which is due to the principles of Java HotSwap (remember that method body redefinitions would not affect methods active on the stack at the moment of redefinition). In our example (see right side of Figure 15), method `measureTemp` (Lines 32–37) is already redefined and thus refers to an object of up-to-date class version `TempSensor_v2`, whereas method `displayTemp` (Lines 24–30) is still active on the stack with the old method body referring to outdated `TempSensor ts`. What appears to be the problem here is that method `notify` (Figure 15, Line 35) would not activate the thread executing method `displayTemp` because method `notify` is executed on a different object. In other words, we have a deadlock.

To prevent deadlocks in multi-threaded applications such as sketched earlier, Gregersen proposes the usage of special synchronization objects, which could be shared beyond different class versions [27]. Figure 16 shows how those synchronization objects could be applied to JAVADAPTOR. Here, class `TempSensor` gets an additional field `syncObj` of type `Object`, which, instead of the `TempSensor` object itself, is used for synchronization (see Figure 16, Lines 6 and 14). If the application must be updated and again the necessary method body redefinitions take effect for one method (in our example for method `measureTemp`, see Figure 16, Lines 32–37), but not for the other (i.e., method `displayTemp`, Figure 16, Lines 24–30), no deadlock occurs. This is because the outdated object (here of type `TempSensor`) and its up-to-date counterpart (in our example an object of type `TempSensor_v2`) share the same synchronization object (i.e., object `syncObj`).

7.2.2. State-loss prevention. Another shortcoming of our current JAVADAPTOR implementation is that it may cause program inconsistencies because of state losses. To illustrate the problem, we use a slightly different version of our small weather station program to be updated at runtime (see Figure 17). Here, we again have the situation that for one method (i.e., method `measureTemp`) the necessary method body redefinition through Java HotSwap took effect, whereas the other method (in our example method `displayTemp`) is still active on the stack with the old method body. Now it could be the case that the outdated method remains active on the stack whereas the state of the referred outdated object (in our case the `TempSensor` object referred by `ts`) is already mapped

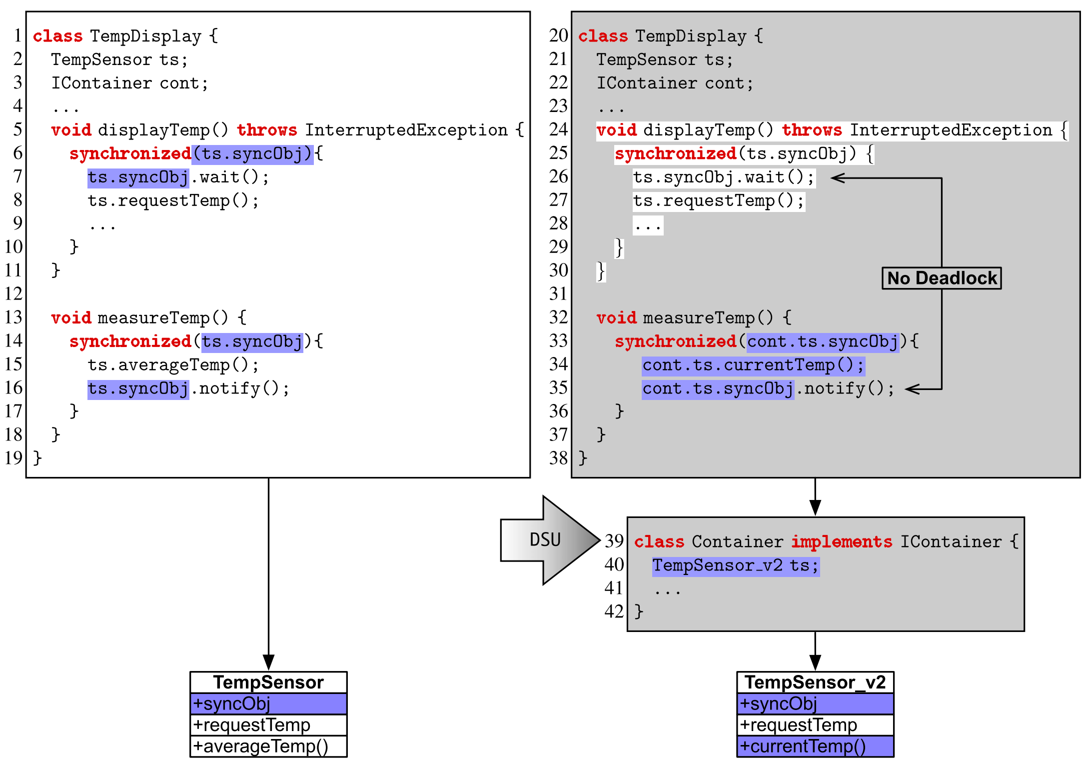

Figure 16. Deadlock prevention through shared synchronization objects.

DOI: 10.1002/spe

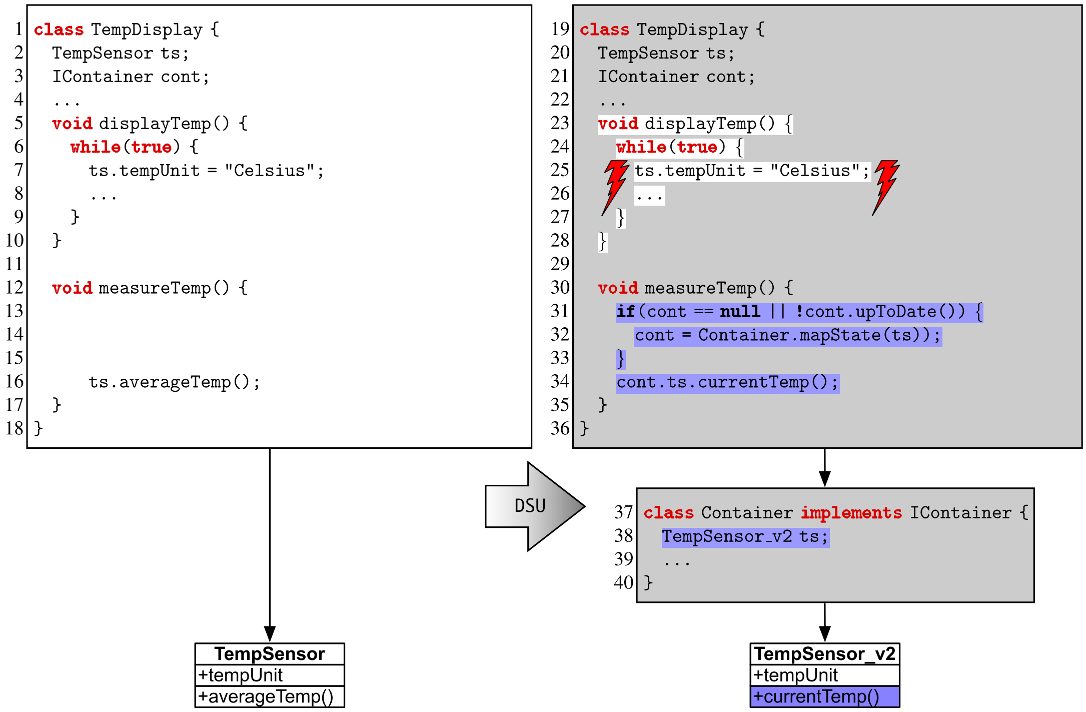

Figure 17. State losses because of dynamic software updates.

to an object of the new class version (here of type `TempSensor_v2`), because another thread executed the redefined method including the state mapping related code (see Figure 17, method `measureTemp`, Lines 31–33). The problem is, that the still active outdated method may change the state of the outdated referred object (such as sketched in Line 25 of Figure 17) and because the state transfer already happened, those state changes would be lost on the new object.

What solves the problem depicted in Figure 17 is to intercept the access to an outdated object and to redirect this access to the corresponding up-to-date object. The challenge is, that the interception and redirection of direct object accesses (such as depicted in Line 25 of Figure 17) is not possible, because of the missing indirection between caller and callee required to hook into the access path. The solution for this problem is delivered by Fowler [25] who argues that, compared with direct accesses, getter and setter methods allow us to flexibly manage accesses to objects.

Figure 18 shows how we plan to use getter and setter methods to prevent state losses because of redefinitions of active methods. Here, again method `displayTemp` scheduled for redefinition is active on the stack (see Lines 43–48 of Figure 18), whereas the redefinition of method `measureTemp` already took effect (Lines 50–56). Only difference to the example depicted in Figure 17 is, that we now access all objects, especially the outdated object of type `TempSensor` referenced by field `ts`, via getter and setter methods (e.g., see Line 45 of Figure 18). To redirect object accesses from within outdated active methods to the up-to-date object, we redefine all methods of old class versions (in our example method `setTempUnit` of old class version `TempSensor`, Line 45) referenced by the outdated method as follows (see Lines 75–81, Figure 18). First of all, we check whether the state mapping already took place (Line 76), for example, because of the execution of an up-to-date method (such as in our example method `measureTemp`, see Lines 51–54). In case the state mapping is pending, we process the state mapping (Line 77). Next, we couple the outdated and the up-to-date object by assigning the up-to-date object to a field of the outdated object (Line 78). Note that the field refers to the same object as the applied container, which ensures that outdated active method as well as up-to-date method access the same object. Finally, we forward the method call to the method of the up-to-date object (Line 80). After this is done, every access to a field of an outdated object from within an active outdated method (e.g.,

```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() {
    while(true) {
      getTS().setTempUnit("Celsius");
      ...
    }
  }

  void measureTemp() {




    getTS().averageTemp();
  }

  //getter and setter
  TempSensor getTS() {
    return ts;
  }
  ...
}
```

DSU

```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() {
    while(true) {
      getTS().setTempUnit("Celsius");
      ...
    }
  }

  void measureTemp() {
    if(getCont() == null || !getCont().upToDate()) {
      setCont(Container.mapState(getTS()));
      getTS().setNewTS(getCont().getTS());
    }
    getCont().getTS().currentTemp();
  }

  //getter and setter
  TempSensor getTS() {
    return ts;
  }
  ...
}
```

```
class Container implements IContainer {
  TempSensor_v2 ts;
  ...
  //getter and setter
  ...
}
```

```
class TempSensor {
  String tempUnit;
  Object newTS;
  ...
  // getter and setter
  void setTempUnit(String unit) {



    tempUnit = unit;
  }
  ...
}
```

DSU

```
class TempSensor {
  String tempUnit;
  Object newTS;
  ...
  //getter and setter
  void setTempUnit(String unit) {
    if(newTS == null || !newTS.upToDate()) {
      //map state from this -> up-to-date object
      //assign up-to-date object to newTS
    }
    ((TempSensor_v2) newTS).setTempUnit(unit);
  }
  ...
}
```

```
class TempSensor_v2 {
  ...
  // getter and setter
  void setTempUnit(String unit) {
    tempUnit = unit;
  }
  ...
}
```

Figure 18. State-loss prevention.

see Access 1, Figure 18) will be redirected to the corresponding up-to-date object (such as through Access 2 shown in Figure 18), and no state will be lost.

7.2.3. Handling of binary-incompatible updates. So far, we discussed how getter and setter methods in conjunction with redefinitions of methods of outdated class version can help us to prevent state losses because of active methods scheduled for redefinition. But, getters, setters, and redefinitions of old methods could do a lot more for us. Coming back to our motivating example, where we are going to remove method `averageTemp` by method `currentTemp` and therefore have to

DOI: 10.1002/spe

replace class `TempSensor` and update calling class `TempDisplay`, conflicts such as depicted in Figure 19 can occur. As in the previous examples, method `displayTemp` to be redefined is active on the stack with the old method body. The problem here is that the method continues to call method `averageTemp` even if this method is removed in new class version `TempSensor_v2`, which is referred to as a *binary-incompatible update* [15].

Currently, we allow caller related methods such as method `displayTemp` to refer to removed methods, fields, or super types, which is no big deal as long as those accesses are read only and thus do not result in program state changes. However, read only accesses may be the exception and methods such as removed method `averageTemp` may alter the program state, which possibly results in wrong program behavior (e.g., method `averageTemp` could overwrite the temperature computed by up-to-date method `currentTemp` with average temperatures). To avoid inconsistencies because of binary-incompatible updates, we must somehow invalidate accesses to removed methods, fields, and super types.

Figure 20 shows how we intend to invalidate accesses to the removed elements. Similar to state-loss prevention purposes, we redefine the methods within the old class versions. What is different is that we do not add state mapping code and forward the calls to the up-to-date class version. We simply remove the original method bodies and corresponding to whether the removed element is a field or a method, throw `NoSuchMethodError` (such as in our example depicted in Figure 20, Line 57) or `NoSuchFieldError`, which does not cause unwanted program state changes and thus has no influence on the program's consistency.

7.2.4. Reflection support. We do not only focus on improved update speeds, thread-safe updates, state-loss prevention, and the handling of binary-incompatible updates. Additionally, we are working on solutions to overcome several problems the different versions of a class present in the JVM may cause. The main issue to overcome is the limited support of our current JAVADAPTOR implementation for reflective calls of reloaded (updated) classes. Under certain conditions, those calls may address old versions of a reloaded class and not the latest class version, which may result in wrong program behavior. This would be, for instance, the case when the class object of the class to be reloaded was cached before the update. Each reflective call based on this cached class object would access the old class version.

With our solutions for state-loss prevention and binary-incompatible updates, which basically forward all requests (including the reflective ones) to the most recent version of a class/instance, we already cover many different kinds of reflective requests. What the approaches do not yet fully cover are string-based reflective calls in combination with type checks (e.g., via `instanceof`).

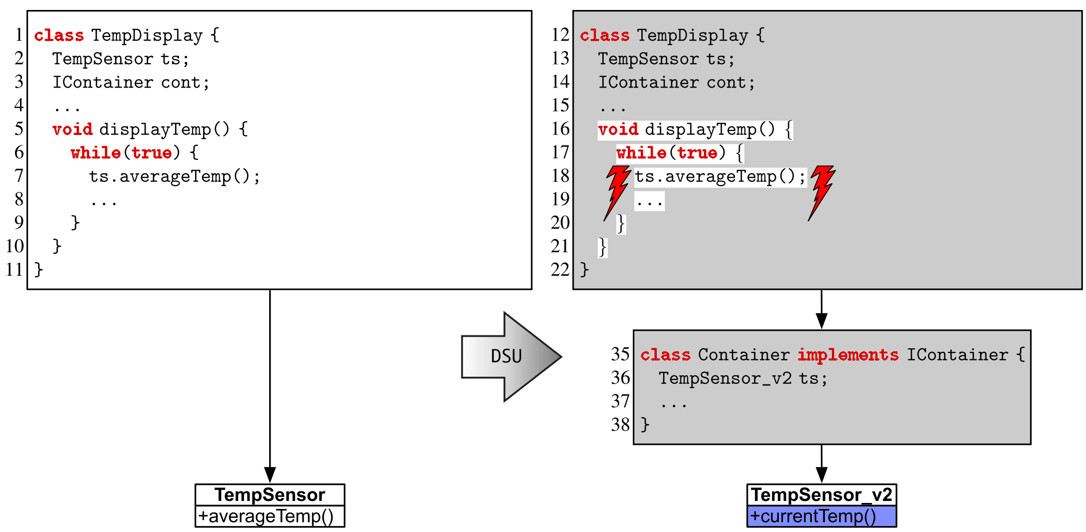

Figure 19. Binary-incompatible updates.

DOI: 10.1002/spe

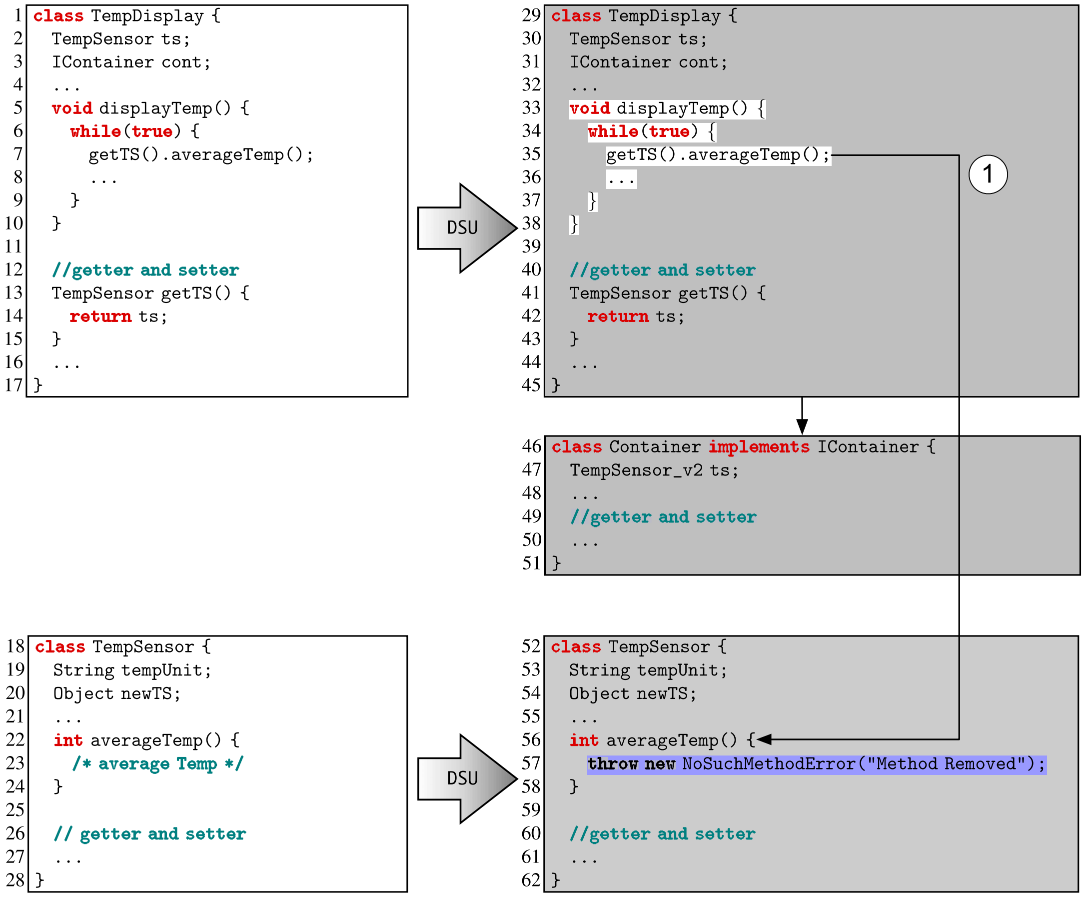

Figure 20. Support for binary-incompatible updates.

Those calls could be supported with two different strategies. First, we could modify the Reflection API in such a way that it redirects even string-based reflective calls to the most recent class version. Second, we could parse the class files for occurrences of string-based reflective calls and change the strings representing a class name to the up-to-date class name. However, further investigations are necessary to find an optimal solution for the described problem.

7.3. *Long-term objectives*

So far, we discussed solutions for issues already solved by other DSU approaches such as Kim's proxy-based DSU approach [14] and Javeleon [27]. What remains an open question to the whole research community is, how to reliably (immediately) apply updates and fully ensure program consistency beyond the updates. Gupta *et al.* stated in [41] that the consistency problem is undecidable. Nevertheless, many related work exists facing the problem (see [26, 42–49]). But, to our best knowledge, some approaches provide approximated solutions only, whereas others are not applicable in real-world scenarios (e.g., owing to the lack of tool support, etc.) or may reject the scheduled update. That is, our big goal with JAVADAPTOR to provide an update mechanism, which fully ensures program consistency, is useful in practice, and reliably applies updates.

7.4. *Discussion*

When looking at the enhancements, we are going to integrate into JAVADAPTOR, one may wonder if those enhancements would compromise one of the contributions of JAVADAPTOR claimed in this paper, for example, its performance. Particularly, the system-wide usage of getter and setter

DOI: 10.1002/spe

methods (note that the getters and setters have to be created for all class and instance fields of all classes including the system classes of Java) would probably cause significant performance penalties. But, contrary to expectations, first benchmark results show that this is virtually not the case, which is because of the excellent optimization capabilities of the JVM and its just-in-time compiler (we found that the JVM is able to optimize getter-and-setter-based field accesses to such an extent, that they are as fast as direct field accesses). In addition, other DSU approaches such as Kim's proxy-based DSU approach [14] and Javeleon [27], which are based on lazy state mappings and use system-wide getter and setter methods for similar purposes as we will do, show that those kinds of enhancements must not cause significant performance drops. For instance Gregersen estimates in [50] the performance overhead of Javeleon at moderate 15 %.

All in all, we are optimistic to provide a stable version of JAVADAPTOR with fast and thread-safe updates, improved state-loss prevention, optimized handling of binary-incompatible updates, and better support for reflective calls, soon. As already mentioned, preliminary results of experiments with JAVADAPTOR prototypes suggest that the planned enhancements must not heavily compromise the performance of the updated program. Another fact that makes us confident to fit JAVADAPTOR with high quality solutions for the mentioned issues is, that we can (to some extent) build on solutions of related DSU approaches (such as presented in [14] and [27]) facing similar problems.

8. CONCLUSION

Dynamic software updates are an often requested approach to update applications while improving the user experience and avoiding down times. Furthermore, DSU supports the software developers because they do not need to restart their applications to test the changed program parts.

However, different from dynamic languages, native DSU support for Java is severely limited. Thus, approaches are needed that overcome Java's limitations regarding dynamic software updates. In Section 1 and 6, we argue that a DSU approach should provide *flexible* runtime program updates without serious *performance* drops. Additionally, it should be *platform independent* and should not dictate the *program architecture*. With JAVADAPTOR, we overcome Java's limited runtime update support and add the runtime update capabilities known from dynamic languages to Java. Furthermore, JAVADAPTOR is (to our best knowledge) the first approach that fulfills all proposed quality criteria: it is flexible, runs on every major (unmodified) JVM, performs well, and does not dictate the architecture of the program. Conceptually, it combines schema changing class replacements with class renaming and caller updates based on Java HotSwap with the help of containers and proxies.

With different non-trivial case studies, we have demonstrated that JAVADAPTOR fits runtime updates of real-world applications executed under real-world conditions. Nevertheless, there is still space for improvements. Currently we are working on the integration of the improvements to JAVADAPTOR described in Section 7, which tackle some issues of the current JAVADAPTOR implementation. However, in the long run, we will focus on the development of solutions to be integrated into JAVADAPTOR that fully ensure the program consistency in the presence of immediate runtime updates, which is still not possible with any existing DSU approach applicable in practice.

ACKNOWLEDGEMENT

We would like to thank Shigeru Chiba for providing the invaluable bytecode modification tool Javassist. Furthermore, we thank Janet Feigenspan for calculating the statistical significance of our benchmark results. Mario Pukall's work is part of the RAMSES project[§§], which is funded by DFG (Project SA 465/31-2). Kästner's work is supported in part by the European Union (ERC grant ScalPL #203099).

REFERENCES

1. Bracha G. Objects as software services, 2005. Invited talk at the International Conference on Object-Oriented Programming, Systems, Languages, and Applications.

[§§]http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/index.htm

DOI: 10.1002/spe

2. Fulgham B, Gouy I. The computer language benchmarks game, December 2011. Available at: http://shootout.alioth.debian.org/.
3. Chiba S, Nishizawa M. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the International Conference on Generative Programming and Component Engineering*. Springer: Berlin, Germany, 2003; 364–376.
4. Chiba S. Load-time structural reflection in Java. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer: Berlin, Germany, 2000; 313–336.
5. Dahm M. Byte code engineering. In *Java-informations-tage*. Springer-Verlag: Berlin, Germany, 1999; 1–11.
6. Haupt M. Virtual machine support for aspect-oriented programming languages. *PhD Thesis*, Software Technology Group, Darmstadt University of Technology, 2006.
7. Tanter É, Noyé J, Caromel D, Cointe P. Partial behavioral reflection: spatial and temporal selection of reification. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Nantes, France, 2003; 27–46.
8. Nicoara A, Alonso G, Roscoe T. Controlled, systematic, and efficient code replacement for running Java programs. *Proceedings of the EuroSys Conference*, Glasgow, Scotland, 2008; 233–246.
9. Orso A, Rao A, Harrold M. A technique for dynamic updating of Java software. In *Proceedings of the International Conference on Software Maintenance*. IEEE: New York, USA, 2002; 649–658.
10. Bonér J. What are the key issues for commercial AOP use: how does AspectWerkz address them? *Proceedings of the International Conference on Aspect-Oriented Software Development*, Potsdam, Germany, 2004; 1–2.
11. Sato Y, Chiba S, Tatsubori M. A selective, just-in-time aspect weaver. *Proceedings of the International Conference on Generative Programming and Component Engineering*, Dresden, Germany, 2003; 189–208.
12. Vanderperren W, Suvee D. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. *Proceedings of the AOSD Workshop on Dynamic Aspects*, Potsdam, Germany, 2004; 120–134.
13. Kabanov J. JRebel tool demo. *Proceedings of the Workshop on Bytecode Semantics*, Paphos, Cyprus, 2010; 1–6.
14. Kim DK. Applying dynamic software updates to computationally-intensive applications. *PhD Thesis*, Virginia Polytechnic Institute and State University, 2009.
15. Gosling J, Joy B, Steele G, Bracha G. *Java(TM) Language Specification*, (3rd Edition). Addison-Wesley: Munich, Germany, 2005.
16. Venners B. *Inside the Java 2 Virtual Machine*. Computing McGraw-Hill: New York, USA, 2000.
17. Lindholm T, Yellin F. *The Java Virtual Machine Specification—Second Edition*. Prentice Hall: New Jersey, USA, 1999.
18. Dmitriev M. Safe class and data evolution in large and long-lived Java applications. *Ph.D. Thesis*, University of Glasgow, 2001.
19. Oracle. Java virtual machine tool interface version 1.2, December 2011. Available at: http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html.
20. Oracle. Java platform debugger architecture, December 2011. Available at: http://download.oracle.com/javase/6/docs/technotes/guides/jpda/.
21. Pukall M, Grebhahn A, Schröter R, Kästner C, Cazzola W, Götz S. JavAdaptor: unrestricted dynamic software updates for java. In *Proceedings of the International Conference on Software Engineering*. ACM: New York, USA, 2011; 989–991.
22. Lieberman H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 1986; 214–223.
23. Opdyke WF, Johnson RE. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*. ACM: New York, USA, 1990; 145–161.
24. Dig D, Johnson R. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 2006; **18**:83–107.
25. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Munich, Germany, 2006.
26. Würthinger T. Dynamic code evolution for Java. *PhD Thesis*, Johannes Kepler University Linz, 2011.
27. Gregersen AR. Extending netbeans with dynamic update of active modules. *PhD Thesis*, University of Southern Denmark, 2010.
28. Malabarba S, Pandey R, Gragg J, Barr E, Barnes JF. Runtime support for type-safe dynamic Java classes. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer: Berlin, Germany, 2000; 337–361.
29. Ritzau T, Andersson J. Dynamic deployment of Java applications. *Proceedings of Java for Embedded Systems Workshop*, New York, USA, 2000; 1–9.
30. Subramanian S, Hicks M, McKinley KS. Dynamic software updates: a VM-centric approach. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM: New York, USA, 2009; 1–12.
31. Liang S, Bracha G. Dynamic class loading in the Java virtual machine. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM: New York, USA, 1998; 36–44.
32. The OSGi Alliance. OSGi service platform core specification, December 2011. Available at: http://www.osgi.org/Download/File?url=/download/r4v42/r4.core.pdf.
33. Oracle. BEA weblogic server using fastswap to minimize redeployment, December 2011. Available at: http://download.oracle.com/docs/cd/E13222_01/wls/essex/TechPreview/pdf/FastSwap.pdf.

34. Zhang S, Huang L. Type-safe dynamic update transaction. In *Proceedings of the Computer Software and Applications Conference*. IEEE: New York, USA, 2007; 335–340.
35. Cazzola W. SmartReflection: efficient introspection in Java. *Journal of Object Technology* 2004; **3**(11):117–132.
36. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley: Munich, Germany, 2004.
37. Pukall M, Kästner C, Saake G. Towards unanticipated runtime adaptation of Java applications. In *Proceedings of the Asia-Pacific Software Engineering Conference*. IEEE: New York, USA, 2008; 85–92.
38. Pawlak R, Duchien L, Florin G, Seinturier L. Dynamic wrappers: handling the composition issue with JAC. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems*. IEEE: New York, USA, 2001; 56–65.
39. Truyen E, Vanhaute B, Joosen W, Verbaeten P, Jørgensen BN. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the International Conference on Software Engineering*. IEEE: New York, USA, 2001; 233–242.
40. Götz S, Pukall M. On performance of delegation in Java. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*. ACM: New York, USA, 2009; 1–6.
41. Gupta D, Jalote P, Barua G. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering* 1996; **22**(2):120–131.
42. Vandewoude Y, Ebraert P, Berbers Y, D'Hondt T. Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering* 2007; **33**(12):856 –868.
43. Kramer J, Magee J. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering* 1990; **16**(11):1293 –1306.
44. Stoyle G, Hicks M, Bierman G, Sewell P, Neamtiu I. Mutatis Mutandis: safe and flexible dynamic software updating. In *Proceedings of the ACM Conference on Principles of Programming Languages*. ACM: New York, USA, 2005; 183–194.
45. Hicks M, Nettles S. Dynamic software updating. *ACM Transactions on Programming Languages and Systems* 2005; **27**(6):1049–1096.
46. Murarka Y, Bellur U, Joshi RK. Safety analysis for dynamic update of object oriented programs. In *Proceedings of the Asia Pacific Software Engineering Conference*. IEEE, 2006; 225–232.
47. Bazzi RA, Makris K, Nayeri P, Shen J. Dynamic software updates: the state mapping problem. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*. ACM: New York, USA, 2009; 7:1–7:2.
48. Karablieh F, Bazzi RA. Heterogeneous checkpointing for multithreaded applications. In *Proceedings of the Symposium on Reliable Distributed Systems*. IEEE: New York, USA, 2002; 140–149.
49. Makris K. Whole-program dynamic software updating. *PhD Thesis*, Arizona State University, 2009.
50. Gregersen AR, Jørgensen BN. Run-time phenomena in dynamic software updating: causes and effects. In *Proceedings of the Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution*. ACM: New York, USA, 2011; 6–15.

DOI: 10.1002/spe