2016 IEEE 27th International Symposium on Software Reliability Engineering

Dodging Unsafe Update Points in Java Dynamic Software Updating Systems

Walter Cazzola
Computer Science Department
Università degli Studi di Milano
Email: cazzola@di.unimi.it

Mehdi Jalili
Computer Science Department
Università degli Studi di Milano
Email: mehdi.jalili@unimi.it

***Abstract*—Dynamic Software Updating (DSU) provides mechanisms to update a program without stopping its execution. An indiscriminate update, that does not consider the current state of the computation, potentially undermines the stability of the running application. To automatically determine a safe moment when to update the running system is still an open problem often neglected from the existing DSU systems. This paper proposes a mechanism to support the choice of a safe update point by marking which point can be considered unsafe and therefore dodged during the update. The method is based on decorating the code with some specific meta-data that can be used to find the right moment to do the update. The proposed approach has been implemented as an external component that can be plugged into every DSU system. The approach is demonstrated on the evolution of the `HSQLDB` system from two distinct versions to their next update.**

***Index Terms*—Dynamic software update, DSU, Dynamic update validation, JavAdaptor**

I. Introduction

Every program needs to evolve in order to fix the bugs or to add new functionality. The typical way to update a program is to stop the running program, modify its code and then restart its updated version. This approach is not always acceptable. Stopping the execution of some kind of programs could cause financial losses or life-threatening risks. Online transaction systems, life-support systems and so on are placed in this category. These types of programs must continuously update without disservices. *Dynamic Software Updating (DSU)* [27] addresses this problem by changing a program at run-time without stopping its execution.

A number of dedicated systems have been developed to support this issue in different aspects [3], [30], [8], [5]. However to investigate this approach in *general-purpose languages (GPLs)* has a special place. Some mechanisms have been proposed to support dynamic updates on programs which are developed in GPLs such as C [20], [17] and Java [31], [28], [26]. These systems provide a code level solution to update a program at run-time. Along with other parameters such as type safety, low-disruption, flexibility and so on, one of the most important concerns with these approaches is when to deploy the update without introducing any fault or unexpected behavior.

Given P_0, a running program, this can be updated to P_1 in two ways: 1) P_0 can be either stopped and a new version P_1 with the needed changes is started instead (*cold restart*) 2) the code of P_0 can be dynamically updated to the new version P_{0u} without stopping (*dynamic update*). To define an equivalence between these two approaches the *reachability* property introduced by Gupta *et al.* [14] can be used. A *dynamic update* P_{0u} for P_0 is equivalent to the update P_1 you get via *cold restart* if and only if after the update, P_{0u} execution eventually reaches some states that P_1 execution would meet. It is formally proved that the reachability is generally undecidable.

Even if the automatic validation of any generic dynamic update is not feasible; it is still possible to bind the update of a program to only those points of its execution that drive to a valid dynamic update. Every program has its own semantics and there is a logical relation between two successive versions of such a program. The program developer is the best person who knows the program semantics and the logical relations between two successive versions of the same as well as which constraints should be respected in order to proceed with the update. Turning one version into another is safe only when the changes to apply respect the imposed constraints. Therefore, for every program a dedicated collection of constraints should be provided and the updating process should verify these constraints before the deployment of the changes. The DSU should be in charge of verifying these constraints before the updating and to subdue the update itself to the result of the verification in order to leave the program stable.

In this paper, we propose an automatic method to determine some points in the program execution, named as *unsafe update points*, that if used to update the running application will drive to an erroneous situation or a failure. These points are constraints that the DSU should dodge when it is deploying the update. To support this idea a collection of meta-data are introduced that can be used to decorate the program code with the constraints that should be verified. It is also presented the algorithm that the updating process should use to validate the changes against the program constraints. The proposed approach is independent from existing DSU methods and can be plugged into any method as a pre-update component called *validator*.

Sect. II describes how a program executes at the dynamic update time and it introduces, through a running example, how a fatal error in a program can occur due to a wrong update point choice. In Sect. III, we describe a possible process that

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.17

permits to automatically determine the unsafe update points in the application execution and mark them as points that the DSU should dodge during the dynamic update. The implementation of this idea is explained in Sect. III-C. Moreover, in Sect. IV, we show how the points marked as unsafe are dodged by the DSU systems with the help of a *validator* component. To demonstrate the feasibility of the proposed approach we used it during the update to their next release of two distinct versions of the HSQLDB system, the results of the experiment are reported and discussed in Sect. V. Sect. VI discusses the related work and finally in Sect. VII, we draw our conclusions.

II. Motivation

Dynamic updating of a running system usually includes two steps: first the update to the code is deployed then the state is migrated from the old version to the new one. Neither the code updating nor the state migration are instantaneous, even if more and more often the state migration is unnecessary, such as in JavAdaptor [26]. Since the changed code is deployed during the system execution, the changes cannot affect the portion of code while in execution but have to wait its reloading. That is, the function/method in execution during the updating will finish its computation with its old implementation; the new implementation will be used only on the next call. Only when the full application is using the new code the updating process can be considered complete.

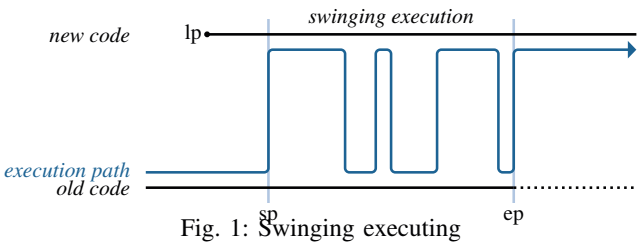

Fig. 1: Swinging executing

As shown in Fig. 1, the code update starts at the time moment labeled with lp, but its effects are disclosed only at the moment labeled with sp. While an application is running the old code, the *DSU system* deploys the new version of the code. DSU systems deploy the new code through several techniques that range from the use of indirection through type wrapping and proxy to the direct injection of the new code. In spite of how the deployment happens, there is always a moment where portions of the old code and new code are alive together. In particular, when the new code is initially loaded into the memory, in the call stack for the current execution there are still portions of the application's old code. However, all the new calls from the old code are directed to the corresponding methods in the new code if any. Due to this reason, the application execution swings between the old and the new version. This situation continues until all of the call frames in the stack pointers refer to the new code and the update process finishes at the moment labeled with ep. After this point, the application only uses the new code and the old code becomes inaccessible. In the period from sp to ep the application execution can manifest some flaws and its state

```
File file;
void action() {
   initialize();
   readFile();
}
void initialize() {
    ...
}
void readFile() {
   file = new File(path);
   FileInputStream inFile =
       new FileInputStream(file);
           ...
}
```

(a) before the update

```
File file;
void action() {
   initialize();
   readFile();
}
void initialize() {
   file = new File(path);
          ...
}
void readFile() {
   FileInputStream inFile =
       new FileInputStream(file);
            ...
}
```

(b) after the update

Listing 1: A Sample Program

is potentially inconsistent. This situation does not occur when either the old code or the new code are run separately. This *transient inconsistency* [12] has been disregarded in some DSU systems [26], [13] because it is considered negligible and in most cases it is automatically called off when the application fully switches to the new code. Such a risk is not always acceptable because the critical application could move to an illegal state and crash or could emit wrong data that cannot be rolled back. On the other side, some DSU systems, as JVolve [28], adopt a conservative policy and start the update deployment when they are sure that no piece of the changed code is still in the call stack. Apart that postponing the update is not always feasible or desirable, this still does not grant the updating process from inconsistencies.

Therefore, the DSU system has to live together with the swinging execution or to stop the application execution to permit a safe update (that by definition is not always possible). The only thing that DSU systems can manipulate is the start of the swinging (sp point in Fig. 1). Starting the update in a safe point, the execution swinging cannot move the system in a *transient inconsistency* state. A *safe update point* [28] can be defined as a moment during the program execution that if DSU system starts the updating process in that moment, the program does not crash nor misbehave. It means that the program can safely be updated without moving in any transient inconsistency state.

Finding a safe update point is a challenge in DSU systems [8], [32] and many DSU approaches neglect this aspect. A safe update point can be determined when i) the state of the running application is available ii) the type of changes is known and iii) it is possible to predict the impact of the change during the dynamic update. To this respect, every execution of any application can have multiple safe update points or none at all.

Let us explain the impact of choosing a wrong update point by an example. Listing 1(a) shows a Java piece of code for processing a file. The `action` method calls the `initialize` and `readFile` methods. The `file` object is created and used by the `readFile` method in the old version. Instead in the new version, Listing 1(b), the `file` object creation is moved to the `initialize` method. Let us consider that the old version

of the application finishes the execution of the `initialize` method (row 3 in Fig. 1(a)) when the new code is replaced. In this case, the call to the `readFile` (row 4 in Fig. 1(a)) method will call the new code where the `file` object is used without a previous initialization; therefore its attempt to access the object will raise an exception and the application will crash. To avoid this situation, the update should be postponed until the execution of the `action` method is completed. In this example the `initialize` and the `readFile` methods are changed in the new version whereas the `action` method is unchanged. In a similar case, also a conservative approach, as the one used in JVolve, would fail since at the moment of the update none of the methods in the call stack are directly involved in the update (the `initialize` has been just popped from the stack and `readFile` has still to be pushed on it) but the inconsistency would occur and it will be due to an unchanged method on the stack at the moment of the update. This situation can only be detected by predicting the impact of a change.

III. Towards a Valid Update

The example of the previous section demonstrates that to update a running application is a delicate matter. The way the update occurs in many DSUs forces some delays in the completion of the effective deployment of the new code and the application temporary runs with a mix of old and new code. A situation that can bring to wrong results and failures.

A valid update occurs when the execution with a mix of old and new code *cannot* drive to a *transient inconsistency* [12]. To automatically determine a point in the application execution that drives to a valid update for any computation without external hints is basically impossible [14]. As discussed in [27], [21], it is instead possible to express constraints about the execution, to mark some points in the execution as unsafe—that is, if the update starts in that point it will drive to a transient inconsistency or to a crash—and to coordinate the update deployment according to this extra information.

The developer knows the application logic and he/she can use some meta-data (in Java parlance: annotations) to provide both the constraints to be validated and the known safe/unsafe update points. These meta-data can be used when the system urges to be updated to both determine a safe moment in time where the update can start and to dodge the points known as unsafe. A component—named *validator*—external to the DSU system will coordinate the update according to the meta-data (details on the process in Sect. IV).

A. Automatic Annotating

The annotating process is not difficult but it is time-consuming and potentially error-prone when manually done. Moreover, since the code by definition is in a continuous evolution also the related annotations should be updated accordingly at every change. These two aspects render preferable to have the code automatically annotated and the annotations automatically maintained. Even if it is unavoidable to have the constraints on the behavior manually specified by the developer, it should be at least possible to determine the unsafe update points that the *validator* should avoid. This work will focus on this last aspect.

The key idea to automatically determine an unsafe update point and then annotate it in the code consists of forecasting how the application would behave if the changes were deployed in a certain moment. Relevant to each simulation scenario are i) the point in code where the update deployment starts, ii) which portion of the old code is active when the update deployment starts (that is, the code that temporarily cannot be updated) and iii) how this code would interact with the new code. If one of the simulations introduces a transient inconsistency or worst it breaks the application execution, the code related to the used scenario is marked accordingly.

Let us consider that an application is executing the method `m` when the DSU system deploys the changes to the application. Since the method `m` is still in execution this will continue to use the old code version but every call it would perform will use the updated code instead. When the new code is deployed the change gradually takes place starting from the methods not in use and keeping those on the call stack unaltered up to when they are popped out from the call stack. So, for example, if the method `m` still in the call stack calls a method which is not in the call stack and that has been removed in the new version, the application will crash. A similar situation could be prevented by analyzing the running code (in our case the new version of the application code plus the old code stuck on the call stack) looking for problems. In this particular case, the problem could also be found by the Java compiler that cannot compile the code with a all to an unimplemented method.

Several tools have been developed that look at an application code for syntactical errors, potential logical errors and warnings. To cite a few, we have FindBugs [16], PMD[1] and JLint [4][2]. All these tools are *static* analyzer tools that *need the full source code available* in order to perform their analysis. In our case the code is the result of a partial update where some portions of the new code live together with the old and still running code. So the code to be analyzed is not available and in particular the portion of old code still running varies according to when the update starts.

B. Calculation of the Unsafe Points

When a change is ready to be deployed, several scenarios can be calculated depending on the current active portion of code—that is, code in the call stack—and by simulating and analyzing the application execution it is possible to determine when it is *unsafe* to deploy the changes. Each scenario should be composed of the new application code plus the old code still active. Unfortunately, from the moment when the new code is ready to be deployed to the moment when the update really starts the application is still in execution and the content of the call stack changes. This renders complicate and time consuming to consider the real content of the call stack in order to calculate the various scenarios and the unsafe update

[1] http://pmd.sourceforge.net
[2] http://artho.com/jlint

points. Rather, it is preferable to widen what can represent a potential risk from the code still in the call stack and cannot be updated to the old code that uses code that has been modified or removed in the new version that if still active could bring to a transient inconsistency. This has the benefit of being dependent only on the old and new version of the code and not on the current execution and on how it evolves; basically making static a dynamic decision process.

A new version of the application code (*new*), is derived from the old version (*old*) by adding some new code (Δnew), by removing some old code now useless (Δold) and by replacing some portion of the old code ($\Delta old'$) with a new variant ($\Delta new'$); ($\Delta new'$) and ($\Delta old'$) shares the same names (methods, classes, ...) but not the same behavior. As in

$$new = old + \Delta new - \Delta old + \Delta new' - \Delta old'.$$

During the deployment, two more factors enter in the equation: the old code that should be replaced (Δold^+) or removed ($\Delta old'^+$) but that cannot be replaced/removed because still active; these are subset respectively of Δold and $\Delta old'$. A particular note should be made for the code that should replace the code still active ($\Delta new'^+$), this is indeed deployed waiting for a full replacement but any new call to one of its operations will use the new version instead of the one still active.[3] So during the deployment the new code is represented by:

$$new = old + \Delta new - \Delta old + \Delta new' - \Delta old' + \mathbf{\Delta old^+ + \Delta old'^+}.$$

Assuming that the new code is correct—that is, it compiles without errors—the code in Δnew do not call code unavailable after the deployment. Therefore the code in Δnew does not represent a potential issue and can be neglected. Similar considerations can be done for the replacement code ($\Delta new'$) and obviously for the removed code ($\Delta old'$ and Δold). A potential problem is instead represented by the old code (Δold^+ and $\Delta old'^+$) still in the system but intended to be replaced/removed instead. This could use some removed code—e.g., a method or a constructor—or another version of the code that has a different behavior than the expected one and that can bring forth to an inconsistency. In formulas:

$$\Delta old^+ \vee \Delta old'^+ \textit{ refers } \Delta old \vee \Delta old'.$$

The *refers*[4] relationship is the one, the execution simulations has to verify in order to find an execution point that could be considered an *unsafe* starting point where to deploy the update.

As for the initial considerations we cannot easily access to the code still active and the calculation of the unsafe points is done statically. To this respect, what we know is the source code of the application before and after the change and consequently the extent of the change itself. The described *refers* relationship therefore must be relaxed to:

$$\Delta old \vee \Delta old' \textit{ refers } \Delta old \vee \Delta old'.$$

Basically, the relationship looks for pieces of code that should not be there if they refer to other pieces of code that should not be there. Please note that not all the old code must be checked because the portion that remains unchanged cannot introduce inconsistencies (reductio ad absurdum, if a piece of code marked as unchanged should refer to a method that does not exist anymore in the new code, this would not compile and would break the initial correctness assumption and it should have been marked as modified instead). A similar consideration can be done for the new code. Moreover, it is possible to limit the check to only the first call of the modified/removed code instead of the whole chain of calls because the next call will use the new code and the risk for a transient error is avoided.

The verification of the *relaxed refers* relationship is pretty straightforward. For every element (classes, methods, constructors, ...) $e \in \Delta old \cup \Delta old'$ the code to be checked will be:

$$new_e = old + \Delta new - \Delta old + \Delta new' - \Delta old' + e.$$

Note that, e is also present in $\Delta new'$ if it belongs to $\Delta old'$ and it should be removed from $\Delta new'$ to avoid a compilation error due to a name clash. The various versions of new_e are then checked for problems (details in Sect. III-C) and when a problem is found the execution of the corresponding e is marked as an unsafe update point.

C. Techinical Details

First of all, the *old* source code is compared against the *new* source code to extract the changes. Under the initial assumption that the *new* code is correct, the novel elements cannot introduce references to removed code and the references to modified code will activate the new version of it. Therefore, the only changes we are considering are the removed code (Δold) and the modified code ($\Delta old'$) from the *old* code. In particular, the considered changes are:

- removed classes and interfaces;
- changed class and interface declarations;
- removed fields;
- changed field declarations;
- removed methods and constructors;
- any change to the method and constructor signatures apart the operation name (that it is considered as a removal plus an addition for a new operation);
- any change to the method and constructor body such as the addition and the removal of statements;

Comparing two source codes is challenging. To have an acceptable result, two *abstract syntax trees* (ASTs) must be compared. Several tools have been developed to extract the differences between two ASTs such as `Change Distiller` [11], `Gum Tree` [10] and `Dependency Finder` [29]. We used `Dependency Finder` because it provides the results in an XML format that we can easily work on in the next steps. Moreover, `Dependency Finder` permits also to correlate the modification in the *old* source with the change in the *new* version and to find any dependency from the changed code to other changed code (the

[3]Note that here we are speaking about the deployed code and not the source code used to do the validation check.

[4]Where with the verb "to refer" we means any use of an element of the set, such as invocation of a method in the set or of a method out of the set but that has an argument of a class in the set.

refers relationship previously introduced). The first kind of correlation is used to calculate the correct variant of the new_e that would consider eventual name clashes. The second kind of correlation permits to limit the number of considered new_e variants to those that effectively could introduce a transient inconsistency.

Once that the set $\Delta old \cup \Delta old'$ has been extracted and the *refers* relationship calculated, the variants of the *new* code (new_e) can be calculated. Several tools have been developed to analyze and transform a source code, such as `RASCAL` [19] and `Spoon` [22]. In particular, `Spoon` has been developed to work on Java source code and therefore better fits our needs. `Spoon` permits to build an in-memory meta-model out of the application source code and provides an API for directly analyzing and modifying a Java application code. From the *old* and the *new* source code, `Spoon` generates two distinct meta-models. Then every element $e \in \Delta old \cup \Delta old'$ belonging also to the domain of the *refers* relationship is added to the meta-model for the *new* version with all the cares about the name clashes. The just built meta-model represents one variant of the *new* code (new_e) that can be compiled in order to determine if it generates an error or some warnings (that still represent potential problems).

To let `Spoon` generates a meta-model and then to variate such a meta-model is faster than generating a temporary source code on the hard disk, compile it and manually check for compilation errors, as reported in [22]. All the generated variants are stored in a database with the corresponding errors and warnings for further analysis. `Spoon` reports the found problems grouped by degrees of importance. Some of them are critical such as correctness and security errors but other problems like bad practices and performance warnings can be ignored since they do not represent an immediate problem. Based on these data, the related elements can be annotated. `Spoon` also supports the annotating of the interested elements and the generation of the new annotated source code. The pseudo-code in Algorithm 1 recaps the whole process.

By using this approach all the critical parts of the application can be extracted. Moreover some warnings can be shown to the developer about the risk of dynamically updating application in some points. However, this is not always sufficient because some changes affect the involved resources instead of the code and the dynamic updating may still fail when these resources are used by the application. Consider, for example, the Listing 1(a), in this case the change could interest the used file instead of the code. In this case, even the presented analysis cannot detect the potential problem but the risk can be limited by the developer which could annotate the use of the file with some constraints, e.g., the check for the file existence, that the *validator* component can consider during the deployment process (details in Sect. IV).

Algorithm 1: Determining and annotating the unsafe update points.

```
{Δold ∪ Δold'} = Dep.Finder extracts changed/removed elements
foreach I_i in {Δold ∪ Δold'} do
    {e_i} = {e_i} + refers I_i
foreach e_i in {e_i} do
    new_i = old + Δnew - Δold + Δnew' - Δold' + e_i
    ΔPS_i = analyze new_i with SPOON
    if ΔPS_i contains a problem then
        mark e_i as unsafe
```

IV. Annotation Driven Validation Process

Once processed the application source code and determined its unsafe update points, this information must be passed to the DSU system and used to coordinate the update. The easiest way is to add meta-data (Java annotations) to the source code and let the DSU system process them. Java annotations can be processed at compile time to generate some other information as well as they can be accessed at run-time through the Java reflection library. This second possibility is particularly useful in the case of DSU systems where the whole validation should occur at run-time. We use Java annotation because it is a standard facility of Java as well as developers are more familiar with it. Limits and advantages of the Java annotation facility can be read in [7].

Annotations can be easily used to mark automatically determined unsafe update points as explained in Sect. III but they can also be used to provide the developer with a way to manually express constraints on the semantic of the application. For example, let us consider the scenario where the code in Listing 1(a) is left unchanged but the used resource (a file) is changed after the application starts to read it. In this case, the dynamic update should be postponed until the lock on the file is released. Constraints can either express static properties or depend on the application's state and therefore being checkable only at run-time. As an instance of this second case, let us suppose that there is a field in a program used to count the number of connections to the application and that according to some safety policies, the application updating can take place only when the number of connections is zero.

The proposed approach relies on the following set of annotations. All of them are run-time annotations (that is, `RetentionPolicy.RUNTIME` should be set) and the *validator* will use them during the application execution. Details of these annotations are as follows.

`@DSUAtomic`. This annotation can be used to decorate both method and class declarations. In the former case, the method is marked as atomic and all the methods called from its code should belong to the same code version. That is, the updater should take care about active code and has to wait that marked method is removed from the call stack before proceeding with the update. In the latter case, all methods of the marked class are defined as atomic; this is equivalent to annotate every method with the `@DSUAtomic` annotation. This annotation is used for annotating those methods that manifest a potential unsafe update point in their execution.

`@DSUPoint`. This annotation explicitly defines an update point. The updater can proceed with the update without any risk for the application stability when the annotated point is

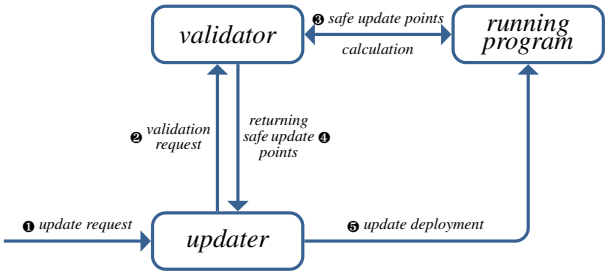

Fig. 2: Dynamic Updating with Validation

reached.

`@DSUConstraint`. This annotation permits to specify a constraint which should be evaluated at run-time. This annotation decorates a class with a constraint that should be respected in order to proceed with the update. The constraint is a boolean expression on the class fields.

All of the annotations have a parameter to support multi threading. By default, all threads of the program are affected by these annotations but developer can change this behavior and specify which threads should be affected.

A. The Validator Component

Existing DSU systems have a direct approach to software updating. The DSU system, identified by the term *updater* in Fig. 2, gets the requested updates (step 1 in Fig. 2) and directly deploys them on the running software (step 5 in Fig. 2) with little or no consideration for the best moment when to do the update. The *validator* component intercepts the requests for an update (step 2 in Fig. 2), calculates the safe update point thanks to the annotations in the code (step 3 in Fig. 2), if the safe update point is determined in a certain amount of time it asks the updater for deploying the changes otherwise the updater is notified that no valid point can be found and the changes discarded (step 4 in Fig. 2). The validator component is external to and independent of the adopted DSU system and can be used with any of them.

Algorithm 2: Main algorithm to find a safe update point

```
connect to the target JVM
foreach program threads do
    suspend it
    if there is a @DSUPoint annotation then
        wait to reach to the annotated point
    else
        repeat
            wait the atomic methods and statements have finished
              their execution
            check for all the provided constraints any time a
              candidate update point is found
        until an update point is found or the timeout is reached
if not timeout then
    command the DSU to do the update
else
    notify the DSU to discard the changes
```

Algorithm 2 reports the pseudo code describing the behavior of the validator component. Before starting the validation process, the validator reflectively extracts from the application all the meta-data about atomic methods and classes. A particular treatment is reserved to the `@DSUConstraint` annotations. A boolean expression is passed to this annotation as a string. Out of this string a new method that simply returns the evaluation of the boolean expression is automatically built and injected in the class annotated with the `@DSUConstraint` annotation by using Javassist [9]. To avoid name clashes the new methods are all named after the annotation name and progressively numbered. The Java reflection library allows the validator to retrieve the methods when needed.

The validator communicates with the other components via the Java Debugger Interface (JDI). JDI provides some APIs to connect to a running JVM launched in debugging mode and to control the execution of the program threads. When an update to the running program is ready, the validator reads the JVM threads list and finds target program threads and suspend one of them. The validator looks for `@DSUPoint` annotations in the program classes. If it finds such an annotation, a breakpoint is set on the annotated expression/statement and the program execution is resumed until it reaches the breakpoint. Such a point has been explicitly marked as a safe update point by the programmer and, now, when the execution reaches the breakpoint the program is stopped again and the updater is asked for deploying the changes. If there is not any `@DSUPoint` annotation the validator tries to find a safe update point by using the other available meta-data. In particular, the validator analyzes the call stack looking for the first frame related to a method either defined as atomic or belonging to a class defined as atomic. If it finds a frame about one of these cases, the analysis is stopped, an event request for this frame is set and the program thread is resumed. The program continues its execution until the marked frame is removed from the call stack. After this method is removed from the call stack the thread is suspended again and we are sure that no other atomic method is in the call stack because the frames still in the stacks have already been checked. At this point, all the active objects are examined to check if their classes have `@DSUConstraint` annotations. In this case, the correspondent methods are invoked to check if the system passes the constraints. If not all constraints are satisfied, the validator put a watch point on the fields involved by the constraint. The thread is resumed again and suspended when the watch point is triggered by a change to the monitored fields. At this point the check starts again from the beginning. Only when the requirements on atomicity and the imposed constraints are satisfied the updater is informed and the changes are deployed on the running application. This process is repeated for every program threads. The search for a safe update point could take too long to be satisfied and therefore a timeout is set; in this case the validator informs the updater that the update cannot be deployed and the changes are discarded.

TABLE I: Details about the two versions of HSQLDB.

Version	1.8.0.9	2.3.2
Number of Classes	363	675
Number of Methods	4700	10037
Number of Constructors	490	965
Number of Fields	3861	7937

V. Demonstration Study

In order to demonstrate the presented approach, we considered two versions (1.8.0.9 and 2.3.2) of the Hyper SQL Database (HSQLDB)[5] and their dynamic evolution to the next version; similarly to what has been done in [26]. In this experiment the presented approach is used to find the unsafe update points and improve the possibility of the DSU system of dodging these critical points during the update deployment. The experiment has been replicated on two versions of the HSQLDB application to widen the variance in the update situations and to analyze how our approach behaves when the application size grows. Table I reports some data about the two considered versions.

Dependency Finder has been used to extract the changes from one of the considered versions to the next version. Table II reports the found changes for the shift from version 1.8.0.9 to 1.8.0.10 and from 2.3.2 to 2.3.3 in the second and fifth column respectively. As previously discussed, our analysis focused only on the modification and removal of elements; new items cannot access to the old code nor be accessed from the old code. So the elements introduced in the new version are irrelevant from the standpoint of the calculation of the unsafe update points and not extracted by Dependency Finder. Then for every extracted element, the elements (mainly methods and constructors) that refer to it are extracted as well. The number of reference to each kind of change is reported in the column labeled with «References» of Table II (the third column for version 1.8.0.9 and the sixth column for the 2.3.2). Please note that an element can use more than one changed element and the reported references reflect this fact. The number of distinct elements referring to the modified portion of the code is reported in the last row of the table.

Then all the elements (methods and constructors) that refer to a changed element are checked. According to the approach presented in Sect. III this implies the construction of several variants of the new code each of them enriched with one of these elements. In the case of methods, the old version is picked up from the source code and added to the new code within the corresponding class. To avoid *duplicate name error* at compile time, there are two possibilities either the name of the introduced method is changed or the corresponding method is removed from the new source code. But if we would remove the new version of the method, any recursive calls should not activate the new version of the code violating the principle that every new calls should refer to the new code version. So the new version of the method must be kept and we changed the name of the added method. The new name is irrelevant since it cannot be called by the new code and it can be changed to any arbitrary name not already in the target class. A different situation arises for the constructors since their name cannot differ from the name of the host class but on the other side a constructor is never recursively called. So, the old version of a constructor simply replaces the new version in the corresponding class.

Spoon permits to modify the application meta-model according to needed changes and then it can generate the corresponding temporary code ready for the analysis. Spoon can detect 626 different kinds of potential problems divided in 16 categories. Five of these categories represents the code situations that drive forth to a run-time fatal errors when executed; they are labeled as: IMPORT, INTERNAL, MEMBER, SYNTAX, and TYPE. We used Spoon to analyze the temporary versions of the new code generated as described to look for potential problems bound to the added element (as a reminder, the rationale is that the added element represents an old element active during the deployment that remains unaffected by the change). The results of the analysis are reported in Table II in the fourth and seventh columns.

The analysis in our demonstration study is limited to the fatal errors. In our results there are no IMPORT error because changes to the code never involve the import section. Similarly, we do not have any SYNTAX error because both versions of the source code can be compiled without errors and our addition cannot change this. The TYPE kind of errors instead occur when there is a type mismatch. For example, in our study, it occurred when the constructor of the org.hsqldb.server.OdbcPacketOutputStream class has been changed so that instead of an object of type HsqlByteArrayOutputStream it requires an object of type byteArrayOutputStream. When the new version of the constructor is replaced with the old one it hits a compilation error with the message: «Type mismatch: cannot convert from HsqlByteArrayOutputStream to ByteArrayOutputStream» due to the attempt of invoking the constructor of the parent class with the wrong kind of objects. Compare the old code in Listing 2(a) with the new in Listing 2(b).

The MEMBER kind of errors usually occurs when an element tries to access to a removed or modified element. For example, in our study, it occurred when in the new version of the org.hsqldb.persist.RAFile class, the database field has been removed and the old constructor tried to access it and the error «database cannot be resolved or is not a field» occurred. Another example, related to the case of a modified element occurred when the initParams method in the org.hsqldb.persist.Log class tried to access a field which is not visible in the new code. The visibility of propLogSize is changed from public to package and raised the error: «The field Logger.propLogSize is not visible».

The INTERNAL kind of errors is related to fatal problems which could not be addressed by external changes and requires an edit to be addressed. This kind of errors rarely occurred in our study. An occurrence of

[5] http://hsqldb.org/

TABLE II: Changes, references and errors in the HSQLDB demonstration study.

Kind of Changes	HSQLDB 1.8.0.9			HSQLDB 2.3.2		
	Changes to 1.8.0.10	References	Compile errors	Changes to 2.3.3	References	Compile errors
Modified classes and interfaces	1	0	0	4	695	148
Removed classes and interfaces	1	0	0	0	0	0
Changes to the method bodies	53	93	6	779	748	113
Changes to the method signatures	23	38	12	183	127	22
Removed methods	9	22	22	113	153	109
Changes to the constructor bodies	4	9	0	47	22	0
Changes to the constructor signatures	2	0	0	3	0	0
Removed constructors	0	0	0	18	1	1
Modified fields	10	30	12	156	237	38
Removed fields	6	19	19	44	131	116
Sum		163	33		1462	261

```
class OdbcPacketOutputStream extends DataOutputStream {
   private HsqlByteArrayOutputStream byteArrayOutputStream;
           ...
   protected OdbcPacketOutputStream(
      HsqlByteArrayOutputStream byteArrayOutputStream)
      throws IOException {
         super(byteArrayOutputStream);
         this.byteArrayOutputStream = byteArrayOutputStream;
         reset();
   }
           ...
}
```

(a) old code

```
class OdbcPacketOutputStream extends DataOutputStream {
    private ByteArrayOutputStream byteArrayOutputStream;
                ...
    protected OdbcPacketOutputStream(
       ByteArrayOutputStream byteArrayOutputStream)
       throws IOException {
          super(byteArrayOutputStream);
          this.byteArrayOutputStream = byteArrayOutputStream;
          reset();
    }
              ...
}
```

(b) new code

Listing 2: Example for the TYPE problem

this kind occurred in the method getTableSpace in the org.hsqldb.persist.DataSpaceManagerBlocks class. It raised the error message: «The operator >= is undefined for the argument type(s) int, AtomicInteger» due to the fact that originally the field spaceIdSequence was of type int and it could originally be compared with the integer argument spaceId but in the new version its type is changed to AtomicInteger that cannot be compared with an integer through the >= anymore. Compare the old code in Listing 3(a) with the new in Listing 3(b).

As it is reported in Table II, in the study for HSQLDB 1.8.0.9, all the elements (22) that refers to removed methods (9) and constructors (0) raise an error when introduced in the new version. This is not a surprise because these elements do not exist in the new version and if an element try to access these items it will provoke a compilation error. Did not happen the same

```
public TableSpaceManager getTableSpace(int spaceId) {
        ....
    if (spaceId >= spaceIdSequence) {
        spaceIdSequence = spaceId + 1;
    }
        ...
}
```

(a) old code

```
public TableSpaceManager getTableSpace(int spaceId) {
        ...
    if (spaceId >= spaceIdSequence.get()) {
        spaceIdSequence.set((spaceId + 2) & -2);
    }
        ...
}
```

(b) new code

Listing 3: Example for the INTERNAL problem

in the case for HSQLDB 2.3.2. By inspecting the new version of the code, we discovered that some of the methods marked as removed were instead moved up in the inheritance hierarchy. In these cases, any call to this methods from the old code is still valid since the method is inherited by the class that before declared it and the generated temporary code can be compiled without error. For instance, the method addForeignKey in class org.hsqldb.ParserDDL has been moved to the parent-parent class org.hsqldb.ParserTable. So all the calls to this method are valid for both old and new code. Similarly, we also found 11 fields in the class org.hsqldb.navigator.RowSetNavigatorDataTable that moved up into the super class org.hsqldb.navigator.RowSetNavigator. In the program, there are 15 methods which refer to these moved fields; you can find confirmation of this situation from the difference between the references (131) and the detected errors (115) in the *removed fields* row in Table II. When Dependency Finder compares two versions of a class, if the arguments of a method are changed, the method is marked as a removed method and a new method is detected on the new code. This depends on the overriding capability and in some cases such a change does not affect old calls. For instance method getStore in class org.hsqldb.persist.PersistentStoreCollection-

TABLE III: Summary of the results

Program Version	1.8.0.9	2.3.2
Modified code executable	9	186
Modified declaration executable	16	14
Removed executable	6	38
Unchanged executable	2	23
Total number of annotated items	33	261
All the checked executable	163	1460
Percentage of annotated items	20.2%	17.8%

`Database` has a parameter of type `java.lang.Object` which has been changed to `org.hsqldb.TableBase` in the new version. All of the 11 callers of this method in the old source code calls it by an argument of type `org.hsqldb.TableBase`. So the new version of the method is valid also for the old calls. A similar situation occurred for the method `moveDataToSpace` in class `org.hsqldb.persist.RowStoreAVLDisk`. The second parameter of this method has changed from `org.hsqldb.lib.LongLookup` to `org.hsqldb.lib.DoubleIntIndex`. This change cannot affect old calls because `LongLookup` implements the `DoubleIntIndex` interface.

A similar situation occurs also for the modified elements. In Table II is reported that the number of references which represent a problem is less than the total number of changed elements. This is due to the fact that sometimes the change does not affect the general behavior/structure of the application. Several cases can be imagined. For example when the visibility of a field passes from `public` to `package` and it is never used out of the package scope or when a method body is changed but its signature remains unchanged. As shown in the Table II the percentage of referenced elements which represent a problem is lower than the percentage of declaration and removed items. This difference is more sensible in version 1.8.0.9. However it cannot be generalized since it depends to the type of changes.

The summary of the results is shown in Table III. A most important result is related to the unchanged elements. Our experiments shows that the dynamic update of a program can be broken even from inside an unchanged method. Another point is that if a DSU system follows a conservative policy (do not update until there is a modified item or a reference to a modified elements on the stack) the update deployment can be uselessly delayed. In our demonstration study about 163 and 1460 elements should be considered at the update time. Whereas with our analysis it has been reduced to 33 and 261 elements (20.2% and 17.8% of the all checked elements). This represents also the number of unsafe update points we have to annotate with the `@DSUAtomic` annotation and to dodge during the update deployment: a lower number of unsafe update points implies also a more agile update process with less constraint to satisfy.

VI. Related Work

Some of the existing DSU systems in Java do not take care about the safe update points, e.g., JavAdaptor [25], [26], Javeleon [13], FiGA [5], [6] and JRebel [18]. These DSU systems have the *edit and continue* purpose and they are heavily integrated with an IDE such as Eclipse or NetBeans. They are used as a support to development and in these cases to deploy a faultiness update is not as critical as in general since, in the worst situation, the developer can stop the debugging process and start the application again. In this application of DSU systems, the most important parameter is the *wait time* [27]. Developer usually makes a small change on the code and expects to immediately apply it to the running application whereas all the possible validation solutions are time consuming and therefore neglected. In spite of this, some attempts [1], [2] to validate the update have been done in the case of the FiGA DSU framework that tries to verify the logic of the change before the real update; anyway it is still problematic the validation of the update process itself.

Subramanian *et al.* [28], in their JVolve DSU system, impose constraints on the update moment. They do not update deleted or modified methods and some other methods that have been black listed by the users. They prepare a list of modified methods that cannot be updated until they are active. As it has been showed in Sect. V, not all the changed elements represent a problem at update time but the problem can also arise when an unchanged method accesses to modified elements. So limiting the approach to only active changed elements, as seen, is not the right choice.

In the Rubah [24] DSU system, the updates are only deployed at a predefined point of code execution. This point is located in the main loop of the program. The authors assume that the nonstopping systems have an infinite main loop and the best point to deploy the changes is either at the beginning or at the end of one of the stages of such a loop. Reached such a point, Rubah calls a predefined method to check if an update is ready for deployment and when it is the program is suspended to permit the deployment of the update. In a multi-thread application all the threads should reach this *quiescent* point before starting the update phase. Although the authors suggest some mechanisms to prevent the application from blocking, it is difficult to achieve this quiescent situation for all the threads at the same time. Also calling the update function at every stage causes a drop in performance as reported in [24].

Zhao *et al.* [32] performed an exploratory study to find a safe update point. They statically extracted unchanged methods from the classes and initially marked all their lines of code as a candidate update points. Then they reduced the number of these points by sifting them according to three parameters: *timeliness*, *success-rate* and *operability*. They exercise test cases for the old and the new versions of the application and compare execution snapshots. Their mechanism is very time consuming and even their evaluation samples (including only 26 classes) takes more than a week to finish, as reported in the evaluation section of [32]. In spite of that, at the end of the process the safe update point set is still large. Furthermore they suppose the existence of the test cases and that the new version of the application is consistent with its original specification. All constraints quite unrealistic in a real world application.

Hayden *et al.* [15] tries to find a safe update point by

systematic testing. They put the candidate update points before every method calling and test every update point with the program test cases. In this way an enormous numbers of test cases are produced for the program and it takes a long time to exercise all of them. Even if they use a minimization algorithm to reduce the number of test cases.

Tedsuto [23] has been developed to test a program before doing dynamic update. This method uses system-level tests which are provided for both old and new programs to find bugs or misbehaviors that can be induced by the update process. It explores update opportunities during the execution of each test case. This method emphasizes on passing the test cases at the update time while as we demonstrated some fatal errors can still occur regardless of the test cases.

VII. Conclusions

This paper presents an automatic way to determine the unsafe update points to be used in combination with DSU systems, i.e., those points in the application execution that if used as an update starting point will drive forth to a problem in the running application.

The proposed approach is based on the analysis of the new code plus an element from the old version of the code. The idea is that of understanding what would happen if the selected old piece of code is active during the update (and therefore unaffected by the change). If the analysis find out that such a combination could drive to an anomalous situation the piece of code is marked through Java annotations as an unsafe update point. Not all the possible combinations are checked but only those that in some way are interested by the change (either they are part of the change or they uses something that is changed). This significantly reduced the size of the analysis and improves the quality of the results with respect to the other approaches in the literature [15], [28], [32].

A component named validator external to and independent of the DSU systems uses the introduced annotations to dodge the unsafe update points during the deployment of the update. Although this method is time consuming in some cases, we believe that it has some potentiality and can avoid erroneous situations and crashes. The proposed approach is independent of the DSU system and can be easily used with any of the existing DSU systems as discussed.

We demonstrated this approach with the use of the JavAdaptor DSU system [26] on the dynamic evolution of two distinct versions of the HSQLDB application. In the future, we plan to analyze the added overhead and to look for more situations that would need different meta-data to be validated.

References

[1] M. Al-Refai, W. Cazzola, S. Ghosh, and R. France. Using Models to Validate Unanticipated, Fine-Grained Adaptations at Runtime. In *Proc. of HASE'16*, pages 23–30, Orlando, FL, USA, Jan. 2016. IEEE.

[2] M. Al-Refai, S. Ghosh, and W. Cazzola. Model-based Regression Test Selection for Validating Runtime Adaptation of Software Systems. In *Proc. of ICST'16*, pages 288–298, Chicago, IL, USA, Apr. 2016. IEEE.

[3] J. Arnold and M. F. Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. *EuroSys'09*, pp. 187–198, Nuremberg, Germany, Apr. 2009.

[4] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. *ASWEC'01*, pp. 68–75, Canberra, Aug. 2001.

[5] W. Cazzola, N. A. Rossini, M. Al-Refai, and R. B. France. Fine-Grained Software Evolution using UML Activity and Class Models. In *Proc. of MoDELS'13*, LNCS 8107, pp. 271–286, Miami, USA, Oct. 2013. Springer.

[6] W. Cazzola, N. A. Rossini, P. Bennett, S. Pradeep Mandalaparty, and R. B. France. Fine-Grained Semi-Automated Runtime Evolution. In *MoDELS@Run-Time*, LNCS 8378, pages 237–258. Springer, Aug. 2014.

[7] W. Cazzola and E. Vacchi. @Java: Bringing a Richer Annotation Model to Java. *Computer Languages, Systems & Structures*, 40(1):2–18, Apr. 2014.

[8] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *Proc. of ICSE'07*, pages 271–281, Minneapolis, MN, USA, May 2007. IEEE.

[9] S. Chiba. Load-Time Structural Reflection in Java. In *Proc. of ECOOP'00*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.

[10] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-Grained and Accurate Source Code Differencing. In *Proc. of ASE'14*, pages 313–324, Västerås, Sweden, Sept. 2014. IEEE.

[11] H. C. Gall, B. Fluri, and M. Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, Jan.-Feb. 2009.

[12] A. R. Gregersen and B. N. Jørgensen. Run-Time Phenomena in Dynamic Software Updating. *IWPSE-EVOL'11*, pp. 6–15, Szeged, Sept. 2011.

[13] A. R. Gregersen, B. N. Jørgensen, Hadaytullah, and K. Koskimies. Javeleon: An Integrated Platform for Dynamic Software Updating and Its Application in Self-* Systems. *S-CET'12*, pp. 1–9, Xian, May 2012.

[14] D. Gupta, P. Jalote, and G. Barua. A Formal Framework for On-Line Software Version Change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[15] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *Proc. of HotSWUp'09*, Orlando, FL, USA, Oct. 2009.

[16] D. Hovemeyer and W. Pugh. Finding Bugs Is Easy. *ACM Sigplan Notices*, 39(12):92–106, Dec. 2004.

[17] M. Jalili, S. Parsa, and H. Seifzadeh. A Hybrid Model in Dynamic Software Updating for C. *ASEA'09*, pp. 151–159, Jeju Island, Dec. 2009.

[18] J. Kabanov and V. Vene. A Thousand Years of Productivity: The JRebel Story. *Software: Practice and Experience*, 44(1):105–127, Jan. 2014.

[19] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-Programming with Rascal. In *Proc. of GTTSE'09*, LNCS 6491, pages 222–289, Braga, Portugal, July 2009. Springer.

[20] I. Neamtiu and M. Hicks. Safe and Timely Updates to Multi-Threaded Programs. In *Proc. of PLDI'09*, pages 13–24, Dublin, Ireland, June 2009.

[21] A. C. Noubissi, J. Iguchi-Cartigny, and J.-L. Lanet. Hot Updates for Java Based Smart Cards. In *Proc. of HotSWUp'11*, pages 168–173, Hannover, Germany, Apr. 2011. IEEE.

[22] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Software—Practice and Experience*, 2015.

[23] L. Pina and M. Hicks. Tedsuto: A General Framework for Testing Dynamic Software Updates. *ICST'16*, Chicago, IL, USA, Apr. 2016.

[24] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a Stock JVM. *OOPSLA'14*, pp. 103–119, Portland, OR, USA, Oct. 2014. ACM.

[25] M. Pukall, A. Grebhahn, R. Schröter, C. Kästner, W. Cazzola, and S. Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proc. of ICSE'11*, pages 989–991, Waikiki, Honolulu, Hawaii, May 2011. IEEE.

[26] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schöter, and G. Saake. JavAdaptor — Flexible Runtime Updates of Java Applications. *Software—Practice and Experience*, 43(2):153–185, Feb. 2013.

[27] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani. A Survey of Dynamic Software Updating. *Journal of Software: Evolution and Process*, 25(5):535–568, May 2013.

[28] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proc. of PLDI'09*, pages 1–12, Dublin, Ireland, June 2009.

[29] J. Tessier. *The Dependency Finder User Manual*. Directly, San Fransisco, CA, USA, 1.2.1-β4 edition, 2010.

[30] Y. Vandewoude, P. Rigole, D. Urting, and Y. Berbers. Draco: An Adaptive Runtime Environment for Components. Report CW372, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2003.

[31] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic Code Evolution for Java. In *Proc. of PPPJ'10*, pages 10–19, Vienna, Austria, Sept. 2010.

[32] Z. Zhao, X. Ma, C. Xu, and W. Yang. Automated Recommandation of Dynamic Software Update Points: An Exploratory Study. In *Proc. of INTERNETWARE'14*, pages 136–144, Hong Kong, China, Nov. 2014. ACM.