Using Runtime State Analysis to Decide Applicability of Dynamic Software Updates

Oleg Šelajev[1] and Allan Gregersen[2]
[1]*University of Tartu, Faculty of Computer Science, Tartu, Estonia*
[2]*ZeroTurnaround, Tartu, Estonia*

Keywords: Dynamic Software Update, Runtime Phenomena, State Analysis, Reliability, Availability.

Abstract: Updating application code while it is running is a popular approach to the dynamic software update problem. But in many cases the behavior of the updated application bears side effects of the update in the form of a runtime phenomena that breaks application state assumptions leading to unwanted complications. We present a runtime state analysis system, Genrih, that enhances a dynamic system update solution and automatically decides if the state transformation functions of a DSU solution are sufficient for the given update. Genrih analyzes the atomic changes in the updated code compared to the already running version and based on these changes automatically determines whether updating the system's runtime state will lead to the observable runtime phenomena. The designed system does not break the update procedure, but observes the state and produces notifications for enhanced analysis and crash management. The practical evaluation shows that the designed system imposes acceptable overhead and can help the developer be aware of several kinds of runtime phenomena.

1 INTRODUCTION

Software evolves, and existing applications inevitably have to be updated with newer versions. The Dynamic Software Update (DSU) problem deals with updating running software without interrupting their behavior. The general motivation for solving the DSU problem is raised by the fact that mission critical applications cannot tolerate maintenance downtime, which makes updating them non-trivial.

Updating applications while they run is a complex task consisting of two main parts: ensuring that the new behavior that the new version of the program specifies is ready and linked into the running process, and transforming the existing runtime state of the application to accommodate the needs of the new program version.

The first problem has been approached by making the programming languages runtimes aware of possible dynamic updates (Erlang, 2017) (Würthinger, 2010), employing architectures that make dynamic updates easier (Hayden et al., 2012b), or enhancing the existing runtime platform to add the support for dynamic updates (Gregersen and Jørgensen, 2009) (Kabanov and Vene, 2014) dynamically.

The latter problem is a more complex issue, which has been shown (Gupta et al., 1996) to be unsolvable automatically in the general case. Transforming the runtime state to conform to the structure the new version of the program expects is tedious work. The current state-of-the-art way to handle necessary change to the runtime state at the update time is to manually specify state transformation functions that will convert the existing runtime state into a representation suitable for the new version of the program. Since the safety of mission-critical systems is paramount and we do not know how to automate the state transformation fully the DSU systems cannot risk employing automatic state transfer solutions (Hayden, 2012). However, defining these state transformation functions is not easy, time consuming, and error-prone. However, requiring the manual intervention of the programmer is not always appropriate. For example, applying dynamic updates in the development environment to avoid long pauses

In this paper, we limit ourselves to investigating DSU of Java applications. Java represents a popular member of the family of the statically typed object-oriented programming languages and has multiple DSU solutions available.

The current state of the art DSU systems that do not require manual actions from the developers stat-

Šelajev, O. and Gregersen, A.
Using Runtime State Analysis to Decide Applicability of Dynamic Software Updates.
DOI: 10.5220/0006400900380049
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 38-49
ISBN: 978-989-758-262-2

ically analyze the code to determine when it can be updated (Zhao et al., 2014).

In this paper we look at the alternative method for safely updating the Java application without requiring the developer to specify the safe points in the code at which the DSU can update the application without breaking the runtime state.

The goal of this paper is to propose a system that automatically decides at runtime if the automatic state mapping of a given update mechanism is sufficient for the update. The proposed system is orthogonal to the DSU solution used for the updates. The practical evaluation is performed on a prototype which is built with JRebel updating functionality in mind.

2 REQUIRING TRANSFORMATIONS

If we look at the changes that can be induced on Java application classes, some of them can potentially lead to runtime phenomena occurrences if applied without concern, (Gregersen, 2011), (Gregersen and Jørgensen, 2011). However, if the DSU system applies changes deterministically, the outcome of applying the change between the old and the new class is determined only by the DSU and the current runtime state. This means that every change can potentially cause only a limited number of runtime phenomena. For example, the uninitialized field, can only cause *NullPointerException*, which is an example of the absent application state phenomena. It cannot lead to other types of runtime phenomena, like for example "phantom objects" phenomena or any other from the list below. In the paper describing runtime phenomena, (Gregersen and Jørgensen, 2011), Gregersen et al. have mapped certain observed phenomena to the individual class changes that were responsible for them.

Below we list possible changes to the Java code and the related runtime phenomena that we will use throughout this paper. The definitions of the runtime phenomena are taken from the work conducted by Gregersen et al., (Gregersen and Jørgensen, 2011).

Phantom Objects. Are live objects whose classes have been removed or invalidated by a dynamic update, (Gregersen and Jørgensen, 2011). Changes that can introduce phantom objects are for example, removing a class, adding modifier abstract to the class definition, replacing a class with an interface. Indeed, if the update adds a modifier abstract to a definition of a class or replaces it with the interface, it means that in the new version of the application, no instances of that class can be instantiated. Any instances of this class that exist prior the update become phantom objects that have no place in the correctly behaving Java application.

Absent State. Is defined as the situation in which objects or classes having been created in a previous version, once migrated to the new version, lacks a portion of the expected state, (Gregersen and Jørgensen, 2011). Adding an instance or static field added to class, which might not be initialized during the update is probably the most straightforward example of introducing the absent state phenomena. There are several types of changes to the code that can cause an absent state phenomena. For example, adding a new subclass with an intent to differentiate between object using polymorphism can cause it. Since no objects existing prior the update can be of the added subclass, differentiating fails despite the fact it might succeed during a fresh run. If a superclass of a class is changed, the existing objects are not reconstructed, so the fields they extend from the new superclass might not be initialized. Removing modifier static from an inner class means that it gets an implicit out field that should point to an outer class instance. However, it might not be possible to provide it during the update so the value of the field will stay absent.

Lost State. Takes place when an existing state that can be used to differentiate between objects is lost. For example, when an instance field is removed or the type of the field is changed the information held in the field prior to update gets completely inaccessible by the virtue of being removed or overwritten with the default value of the new type. Changes that lead to the loss state phenomena are thus removing instance or static field or changing the field type.

Oblivious Update. Phenomenon is the absence of an expected runtime effect that would have occurred if the system was started from scratch. For example, a change in constructor has no effect on the previously initialized objects, (Gregersen and Jørgensen, 2011). Constructor code change as well as instance or static initializer changes lead to oblivious update phenomena being observable.

There might be other runtime phenomena that are observable after updates, but in this paper we focus on these four phenomena being described and analyzed in the prior work by Gregersen et al. (Gregersen and Jørgensen, 2011). The description of the runtime phenomena given above maps the runtime phenomena to individual changes that cause them. Naturally, if we reverse the mapping, then we can map changes to all potential application level phenomena, these changes can produce if applied by a given DSU solution.

The phenomena listed in the Gregersen's work does not depend on the exact approach used by the

dynamic update system and are solely dependent on the runtime state of the application. These phenomena occur only when the application state satisfies certain conditions, which can be expressed as the queries to the runtime state that evaluate if a phenomena dictated by the changes might be observed. The nondeterminism of the phenomena and their dependency of the runtime state is analyzed in the same work which introduced them, (Gregersen and Jørgensen, 2011).

Here is an example of a query to the runtime phenomena to occur. If a phenomenon reveals itself on the access to the instances of a given class initialized prior the update, the phenomenon is observable if the query "are instances of the class or any of its subclasses present in the application?" returns true given the runtime state at the update moment.

The similar query tells if the following runtime phenomena might be observed: phantom objects because of the class removed, phantom objects because of adding abstract modifier to a class, or changing a class to an interface. Lost state phenomenon is also described by the similar query: "are instances of the class or any of its subclasses present in the application?", however it can be specialized further to analyze if the field of those object instances can be used to differentiate them.

Oblivious update caused by changing the constructor code can be predicted by the same query, however, if it is caused by changing a static initializer of a class, then a simpler query whether the class is loaded into the JVM or not is sufficient, since static initializers are not executed for every object instance.

By describing the queries to determine runtime phenomena, we implicitly presented a mapping between the runtime phenomena and the changes that cause them. The reverse mapping of possible runtime phenomena caused by the changes to the classes of the applications can serve as a good first approximation for the mapping database approach for verifying the update safety. The further work can improve the mappings and include more changes to effects mappings to cover more updates. The exact implementation of the mapping that was used in the evaluation of the approach is discussed further in the paper.

Note, that it is be possible to derive less conservative queries for specific runtime phenomena, but for this paper we used the most straightforward ones, usually saying that the phenomenon is possible to occur if the instances of the changed class exist on the heap.

The approach described in the Static Analysis for Dynamic Updates tool paper, (Oleg Šelajev and Kabanov, 2013) is capable of discovering the atomic changes that were found to be responsible for the above-mentioned phenomena during the experiment. The tool compares application classes one by one and returns the list of individual changes in them. In this paper we present a runtime state analysis engine that provides an insight into the runtime state of the application and is capable of serving the queries mentioned above. This runtime state analysis engine forms the last component of the system required to establish if an update can lead to the observable runtime phenomena.

The dynamic software update solution that is capable of discovering the changes that occurred between the application versions under the update, a database to map these changes to the corresponding runtime phenomena, and an analytic engine to inspect the runtime state can provably establish if the update is safe from introducing these unwanted effects into the updated application.

3 A SYSTEM FOR PREDICTING RUNTIME PHENOMENA

Deciding if the state transformation functions are adequate for a given update is equivalent to analyzing whether the update will produce runtime phenomena. The system implementing the dynamic state analysis to predict if a given update can result in observable runtime phenomena is orthogonal to the actual DSU solution performing the update. The architecture of the integration with DSU solutions is quite straightforward. It requires the access to the original and changed versions of the classes involved in the update before the actual update happening; then it will run the analysis of the application state on the heap and determine if applying the update is safe or if any application level phenomena can be observable afterward.

The system to predict a potential faultiness of a dynamic update consists of the following components: an runtime state analysis engine that is able to respond to queries about the current runtime state; a class diff tool that determines which changes has occurred between two versions of the application classes; a mapping between the changes into the possible runtime phenomena caused by these changes, and a world stopper that can pause the JVM to ensure that the analysis is sound and the application state does not change between the analysis and the application of the update by the DSU solution.

The overall architecture of the designed system, is depicted by Figure 1.

Such a system is developed as a standalone component separate from both application code and the

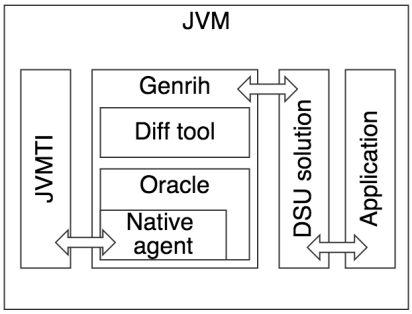

Figure 1: Architechture of runtime analysis system.

DSU solution that performs updates. It embeds both the diff tool to compare class files and the runtime state analysis engine, which uses JVM native agent capabilities to inspect the runtime state. The arrows on the Figure 1, show the main communication patterns between the components. Genrih does not act upon the application itself, but rather integrates with the DSU solution to obtain changed classes and notify about the update safety. The native agent inside the system uses the Java Virtual Machine Tool Interface (JVMTI), (Oracle, 2017) to gain access to the runtime state that is not otherwise possible to obtain from just the Java code. Note that the capabilities of non-native *javaagent* or application level code are not enough to perform the analysis of the Java heap to the extent necessary.

The rest of this section describes responsibilities and a possible implementation of each component and how they all come together to predict the faultiness.

3.1 State Analysis Engine

JVM stores objects on the heap. The memory on the heap is managed: memory for new objects is allocated automatically, reachable objects are often moved around, unreachable instances are garbage collected and the memory claimed by them can be reused.

As we saw above the most of the queries to determine the possibility of a phenomena consist of two questions: 1) is class T loaded into the JVM and 2) are there initialized and not yet garbage collected instances of the class T, sometimes including its subclasses.

Luckily, we can answer both of these questions for a given class T by leveraging the JVM tool interface (JVMTI) and a native agent. JVMTI provides a way to inspect the state of the running JVM program and influence its execution. Determining if a class is loaded in the JVM can be done from the JVM itself, especially if we are not interested in the class loading details and can query just the system class loader. We can use the *findLoadedClass* method of the ClassLoader class to obtain this information.

To count the number of the instances of a class T, we can make use of JVMTI's "iterate through heap" function that takes a class which instances to iterate and a callback function to call for every instance. For our purposes, the callback function just increments a counter for every found instance. If after the invocation of the "iterate through heap" with the given class argument the result is non-zero, there are instances of the given class.

The native agent approach to leverage iterating through heap is general enough to cover more complex queries that might be necessary to introduce for the future enhancements of the system. For example, if one needs to investigate a removal of the field and the possibility of lost state phenomenon, a more complex callback that inspects every instance found by the iterate through heap function to determine if the field in question has a distinguishing value or not. If all the objects have the field initialized to the default value for the field type, removing the field does not actually lose any data.

The discussion section contains more details about which queries were implemented in the prototype of the described system and why.

3.2 Class Diff Tool

The DSU solution must have access to the new versions of the classes to update the definitions in the JVM. We can use a simple cache mechanism to store the bytecode of the loaded classes by the class name. To reduce the memory footprint we can potentially ask the DSU solution if the class in question is reloadable, meaning if there is a possibility that the bytecode for that will change and avoid storing the bytecode for classes that won't be updated. Alternatively, the caching scheme to disk can be employed, which can introduce additional performance overhead, which might not be desirable. However, this becomes an engineering challenge, so for the purposes of this paper we implement the approach that is just functional.

The class diff tool, (Oleg Šelajev and Kabanov, 2013), consumes two sets of classes: old and new; and produces a list of *events* that describe how the classes in the new set differ from their respective counterparts in the old set.

For the class definitions that are different in the old version and the new version there are three main results of the diff analysis: old class does not exist, new class does not exist and both classes exist. First two cases are naturally mapped to the *class_added* and

class_removed changes respectively. If both class definitions exist, class diff tool analyzes them for the differences. The class diff tool described in the Static Analysis for Dynamic Updates paper, (Oleg Šelajev and Kabanov, 2013) does exactly that, it takes two definitions of a class and returns a list of the events: *new_instance_field_added*, *new_static_method* event, etc. Each individual change event has a reference to the class and the class member: method, field, initializer, that were affected changed.

The analysis engine can run the diff tool on every class for which both the old and the new versions are present and obtain the list of exact changes between them. Then these lists of changes are mapped to the possible runtime phenomena they might produce and fed to the runtime state analysis component to determine the possibility of the phenomena given the current runtime state of the application.

3.3 Changes to Phenomena to Runtime State Queries

We need to map the exact changes that occurred between the old and the new versions of the classes involved in the dynamic update. During an update we have the class definitions for both old and new versions of the application. The output of the class diff tool described above is the list of change events that are found between these versions.

In the current work we investigate the following changes that were previously found to be causing the runtime phenomena, (Gregersen and Jørgensen, 2011). The first column of the Table 1 specifies the changes, the second lists the runtime phenomena they can produce, and the third column describes the queries, which reveal whether the phenomena might be observable.

Out of the all changes that can lead to runtime phenomena effects and that are recognized by the system we designed, we handpicked a subset for the showcase. While other changes are also interesting,these have been previously recognized to lead to the runtime phenomena described above, (Gregersen and Jørgensen, 2011).

In the future research the mapping for the changes can be more thorough, however, being able to predict the faultiness with respect to the runtime phenomena observation after the dynamic update containing these changes is a useful result in itself.

The mapping provided above can be directly translated to the code via a series of if-else statements, where the class T is the class currently being diffed. When the queries that correspond to the possibility of observing the runtime phenomena are obtained, the runtimestate analysis engine evaluates them using the queries it knows how to answer. The result shows whether the update is safe from the application level runtime phenomena, with respect to the phenomena and the changes that we analyze.

Table 1: Changes to phenomena to queries table.

Change type	Phenomena	Analysis queries
Class T made abstract change	Phantom objects: instances of class T are on the heap	Class T is loaded and instances of T are on the heap
Class T removed change	Phantom objects: instances of class T or subclasses are on the heap	Class T is loaded and instances of T or subclasses are on the heap
Constructor of class T changed	Oblivious update: existing instances have run previous version of constructor	Class T is loaded and instances of T or subclasses are on the heap
New instance field change in class T	Absent state: old instances of T won't have the new field initialized	Class T is loaded and instances of T or subclasses are on the heap
New static field change in class T	Absent state: static field might not be initialized	Class T is loaded
Static initializer changed in class T	Oblivious update: new version of static initializer is not executed	Class T is loaded
Super class of class T changed	Absent state: instances of T or subclasses might not have field of the new superclass initialized	Class T is loaded and instances of T or subclasses are on the heap or the hierarchy from the new superclass to Object does not declare any fields.
Modifier static removed from inner class T	Absent state: implicit field out on instances of T is not initialized	Class T is loaded and instances of T or subclasses are on the heap

3.4 World Stopper

A JVM embodies a multithreaded environment where different threads, like the garbage collector mutate the global state all the time. The direct consequence of this is that to ensure the soundness of the analysis, we have to synchronize the analysis and the update with the JVM activities external to the DSU solution at hand. One way to do this is to rely on the JVM pausing its work for the internal bookkeeping. However, this might not be utterly portable, so the more direct solution is to use the JVMTI thread suspending functionality and iterate over all threads that are not involved into the dynamic update. Stopping the threads for the analysis bears an obvious performance overhead, which we measure in the practical part of the current research.

Stopping the JVM for the analysis adds the benefit

of knowing exactly what methods are currently active on the stack. If a method body or its signature have been changed in the update and the method is currently running, applying the update can lead to various unwanted side-effects. For example, if the update removed a method which currently executing code tries to call, the best the system can do is to throw a *NoSuchMethodErrors* to communicate inability to locate the method in the updated code. Stopping the world for analysis allows the system to inspect if methods that have been changed are currently active. Such runtime check alone can simplify the type safety of the update process by preventing the updates that modify the currently active methods from being applied. However, in the spirit of not intervening with the update process, one can emit a notification that a currently active method is changed and possible side-effects including among others exceptions about class members not found.

3.5 A Prototype to Enhance JRebel

We implemented a prototype of the system to predict runtime phenomena after update is applied dynamically. We call the prototype Genrih and it is integrated with JRebel dynamic updating functionality.

The general workflow of performing a dynamic update with the runtime state analysis is illustrated in the Figure 2.

The runtime state analysis system marked as Genrih in the Figure 2 receives a request from the DSU solution that an update is available and the list of classes that are going to be involved in the current update. These classes are diffed to obtain the exact changes in the update. The system then stops all the activity in the JVM using the world stopper described above. At this moment no state can be mutated in the application, so we run the analysis of the heap by evaluating the queries that are mapped to the changes. The dynamic update proceeds as follows, if the update is runtime phenomena free at the current point in time, we signal the DSU solution to continue with the update and replace the class definitions involved. After this process finishes, we can resume the paused threads and report that the update has successfully finished.

Otherwise, Genrih still resumes the threads, but schedule the analysis after a small random delay up to 500 ms hoping that the runtime state of the application will have changed by then, removing the objects that are responsible for the possible phenomena after the update. If the following analysis runs show that the state has not changed enough, and the runtime phenomena are still possible, Genrih emits a notification for the developer saying what runtime phenomena is possible and what change is causing it and if known, which objects contain the state leading to the observable difference in behavior.

In the actual system which enhances the existing development time DSU solutions, after showing the notification, Genrih should allow the system to proceed with the update, not to stall the development process.

The exact details of how the DSU solution we integrate with is performing the update are orthogonal to the safety evaluation. Thus, we can treat it as a black box. The only integration points that we are interested in are:

- a signal that the update is available;
- a list of the classes involved in the update;
- the functionality to signal if Genrih determined whether the update is safe to apply.

These requirements are relatively humble, and the integration with an existing DSU solutions is not complex.

Genrih implements the runtime state analysis engine, the mapping functionality of class changes to potential side effects, the integration with JRebel, and the use of a class diff tool.

The next chapter describes the experiment of updating a real-world game application with the runtime analysis system capable of predicting runtime phenomena occurring because of these changes to the code.

4 EXPERIMENT

We have evaluated the designed system on the update scenarios of a Space Invaders game that Gregersen et al. have used to prove the existence of the runtime phenomena, (Gregersen and Jørgensen, 2011). The choice of the application for the experiment is influenced by the lack of the systematic benchmarks or analysis of the development time DSU solutions. Also, reusing a code base that certainly contains several versions of the application different enough to produce runtime phenomena after the updates, is more relevant to the current work than performing the experiment on an arbitrary code.

The performed experiment was designed to provide information about two hypotheses:

1. The system can predict that an update will not cause side effects or provide an immediately qualified feedback regarding possible runtime phenomena.
2. When or if the runtime state at the update time will not lead to the side effects, the system carries

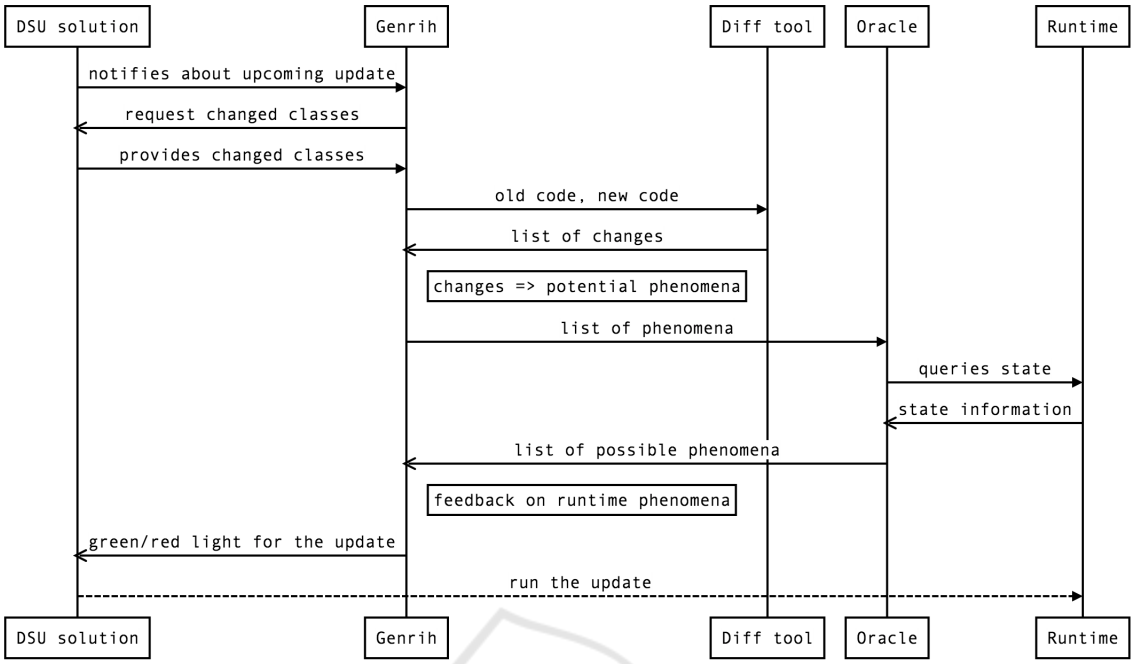

Figure 2: Update process with Genrih.

the update out without the considerable overhead.

The experiment process follows the given procedure. Two versions of the Space Invaders game are manually investigated to find out the changes that correspond to the update. Both versions of the game are started to determine what is the expected behavior of the program.

After some state is reached using the old version of the code, we update the code base to the new version and build the application without stopping the use of it. If the update is successful, we try to observe the runtime phenomena predicted by the manual code analysis. If we observe the side effects, we consult the output of Genrih to verify that the phenomena were predicted, and the notification of its effects is present.

Although the experiment procedure is not automated and relies on manually constructing pre-update runtime state, it does mimic typical application development scenarios, which are the main interest of this work.

The first part of the experiment determines that updating an application with JRebel without considering the runtime phenomena can indeed crash the application. The application under test is the Space Invaders game, the versions of which differ in how they assign the color of the *Shot object*. In the old version, the color is a constant *Color.YELLOW* returned from a *shot.getColor()* method.

In the new version of the code, *Shot* class has an instance field: *Color color* that is initialized to *Color.GREEN* at the end of the only constructor for the class and returned from the getter. The default value for the color field is *null* and all objects initialized within the new version of the game running have the field Initialized during the regular constructor execution. If the color field happens to be *null*, at the moment the redrawing routine asks for it, the *NullPointerException* is thrown and the program crashes. Both versions of the game work if they start from scratch, and the shot objects flying through the screen have correct colors: yellow and green respectively.

When there are no shots visible on the screen when JRebel updates the application, the update succeeds. The shots that are fired afterward are green. However, if the shots are visible on the game field during the update, the game crashes with the *NullPointerException*, because the shots do not have the color initialized.

If this update is triggered with Genrih performing runtime state analysis, it correctly logs the possibility of the **absent state** phenomenon on the pre-update shots instances. This feedback together with the exception stack trace is sufficient to identify the update of the crash reasonably.

The next phase of the experiment involved significantly updating Space Invaders code, going to another major revision of the game. The functional changes in the update add barrier entities that have to be drawn on the field and make the aliens move and shoot back.

Updating from the initial version of the game to

the new shooting version of the game with JRebel brings no visible runtime phenomena. The game proceeds as expected, having the new behavior in place. However, updating the game back to the old version of the code, which has no information about aliens being able to shoot, crashes the program with a *NoSuchMethodError*, because the method *Aliens.fire()*, called from the main game loop is not present anymore.

With the runtime analysis system, the update forward to the shooting version of the code goes in the following fashion. The forward update determines the new instance fields in the *Game* class with the following declarations.

```
private Shots alienShots = new Shots();
private Barriers barriers = new Barriers();
```

The runtime analysis shows that there is one *Game* object on the heap, so the update is postponed due to possible absent state on the Game object after the update. However, since after the update, the application does not crash this false positive feedback is easily ignored.

The downgrade from the shooting version of the game to the basic one crashes with the *NoSuchMethodError* and the notification from Genrih: "Threads are currently running method *Game.gameLoop()* that is changed. Unpredictable update can happen". Together with the *NoSuchMethodError* stack trace, that originates in the *Game.gameLoop()* method, this information is sufficient to consider the DSU being responsible for the crash, not the application logic.

An important additional observation is that, however, most of the time some thread is executing the *Game.gameLoop()* method, there is a small window of time, during the game tick, when it is not on the stack. Then the update proceeds without triggering errors. The dependency of the updates on the current runtime state and given that Genrih can predict if the update will cause no runtime phenomena opens possibilities to stall the update process until the runtime state changes so the runtime phenomena are impossible. We discuss such advancements of Genrih in the discussion section.

The following updates to even more complicated Space Invaders versions occurred in the similar fashion. Without the runtime analysis, adding features and state to the code are handled by existing JRebel well. The downgrades often result in the *NoSuchMethodErrors* caused by calling methods that no longer exist in the new version from the old version of the method still running during the update. Which is the result of not checking the if the updated code is currently actively executed on in the program.

The experiment shows that the designed system provides immediate feedback on the runtime phenomena using an existing stock DSU solution. This feedback and the nature of errors originated in the runtime phenomena clearly indicate that these errors are due to updating the application rather than the code itself. Debugging the issue with such feedback that strongly suggests the origin of these errors in the updating process is more straightforward than without it.

More detailed analysis of the experimental results is presented later in the discussion section.

5 PERFORMANCE ANALYSIS

This chapter describes the performance overhead of automatically determining the applicability of the state transfer functions of a DSU during the dynamic update. The designed system:

- requesting the list of classes involved in the update,
- running the diff on the old and the new version of the classes to find the exact changes,
- querying the runtime state analysis engine about the current runtime state for possible runtime phenomena caused by the changes (for every changed class).

Requesting the changed classes and diffing the result can be done in parallel with running the application so the impact of these actions is negligible and the complexity of the operations is linear in the number of classes changed.

Performing the state analysis is more complex. First of all, it must happen when the application is paused so that the state will remain unchanged between the analysis and the actual moment of the update.

The runtime state analysis engine offers the following API for the queries:

```
boolean isClassLoaded(String className)
boolean hasInstances(String className)
boolean hasFieldInitializer(Class klass,
                            String fieldName)
boolean hasNonDefaultFieldValues(
                           String className,
                           String fieldName)
boolean isMethodRunning(String classname,
                            String methodName)
```

This chapter focuses on analyzing the performance of the implementation of the runtime state analysis engine used in the experiment. The main measurements indicate how much time do individual calls to these methods take and the results can be extrapolated to estimate how much time can a single analysis run take. Given that the number of possible queries are limited and is linear in the number of

changed classes, the extrapolation is straightforward.

The machine where the benchmarks were run has the following configuration: MacBook Pro (Retina, 13-inch, Early 2013) with the 2,6 GHz Intel Core i5 processor, 8 GB 1600 MHz DDR3 memory, and a flash storage hard drive. Java version "1.8.0", Java(TM) SE Runtime Environment (build 1.8.0-b132), Java HotSpot(TM) 64-Bit Server VM (build 25.0-b70, mixed mode) was used to perform the experiments.

The benchmark was run from a JVM process with the heap size of 1GB. During the benchmark, about 70% of the heap was filled with the objects of dynamically generated classes to model the real world performance. We also started 32 background threads to provide the load for the isMethodRunning query comparable to a real world use.

To perform benchmarks we utilized Java Microbenchmark Harness (JMH), (Shipilev, 2017). The benchmark was configured to measure the average time of the execution of an operation based on 10 sample properly warmed runs. Table 2 shows the output of a random run of the benchmark using 10 iterations.

Table 2: Runtime queries benchmark.

Benchmark	Score	Error	Units
isClassLoaded	0.001	± 0.000	ms/op
fieldInitializer	0.761	± 0.062	ms/op
hasInstances	1.430	± 0.343	ms/op
nonDefaultFields	1.513	± 0.306	ms/op
isMethodRunning	0.099	± 0.014	ms/op

The time growth is linear of the number of the instances the system has to traverse. For a million of active object instances, it takes consistently under 60 ms. Varying heap size did not influence the timing on the heap sizes up to 3 GB.

The results suggest that the overhead of running a performance check on an incrementally small update to the application code is sub-second. During the experiment with the Space Invaders game, the updates were postponed by 500 ms if the runtime state does not allow to perform it immediately. The analysis did not noticeably slow down the user experience.

6 DISCUSSION

The current research concentrates on the design of a runtime state analysis system for the JVM that can automatically determine if the state transformation functions of a given DSU solution can satisfy a given update. It does so by analyzing the individual changes that the update consists of and querying the runtime state to verify if the declared capabilities of the DSU can handle the scenario at hand.

The proposed system tackles foremost the updates in the development, where the changes to the application are frequent and typically smaller than the difference between two releases of the application. The main benefit of such system is that the developer after introducing the change would likely run the exact piece of code that was changed to verify the correctness of the introduced functionality. As such, any incompatibilities in the runtime state representations in the old and new code have a much higher chance of being stumbled upon and producing a runtime phenomena of the update.

Sometimes the inability to apply an update dynamically can be noticeable. The incomplete update can lead to application errors or crashes. Others, however, are subtle and have less obvious consequences. The errors require developers to investigate the code base to determine the cause of the occurring behavior. That is counterproductive to the core idea of the DSU for the development environments, which is save time.

In the previous section, we showed a series of dynamic updates of a sample Java application, the Space Invaders game, that illustrated two things. First, the runtime phenomena occur during the dynamic updates, and the updating systems can lack the sophistication to distinguish a safely applicable update from those that will result in a system crash or invalid runtime state of the application. Since the problem of automatic state transformations is not solvable in the general case, (Gupta et al., 1996), any system not requiring manual intervention of the programmer has blind spots for the particular changes and will break the update. This also one can always design an application whose behavior after being updated differs from the behavior of a clean run of the new version. One straightforward way to do this is to rely on the application level data to differentiate between the behaviors. For example, to kill the application process if a certain marker class is already loaded. The old version of the code will then load the class, and proceed to wait for the user input. The new version of the application checks if the marker class is already loaded and decides whether to kill itself. The only way to solve issues like that is to specify the state transformation functions manually. However, it is imperative to avoid the manual intervention from the developer during the updates, since it removes all the time-saving benefits of the dynamic updates.

Second, that our proposed state analysis engine can predict the applicability of the update by query-

ing the runtime state efficiently.

Without the runtime analysis system, even a minor update can result in application crashes as the experiment section showed in the example of dynamic updates applied with JRebel.

The non-deterministic nature of such errors, due to runtime phenomena being dependent on the runtime state at the moment of the update complicates debugging and makes verification of code correctness time-consuming work.

Using the runtime analysis and feedback system described in this paper, can significantly decrease the frustration of encountering unexpected behavior in the updated application.

Indeed, as the experiment showed, we can determine that the state transformation functions do not satisfy the update at hand. Additionally due to the knowledge of what changes the update consists of and which objects are on the heap or active methods on the stack might produce the runtime phenomena, we preventively notify the developer about the upcoming errors, reducing how unexpected these are.

The list of runtime phenomena the system can predict covers the changes to Java programs previously identified to be capable of producing runtime phenomena: adding or removing static and instance fields of the objects, changing actively running methods or the hierarchy of the loaded classes. The updates performed during the experiment show several significant runtime phenomena types and show we can apply the knowledge of the capabilities of the state transformation functions of a particular DSU solution with regards to these.

To utilize the described system with an arbitrary DSU, one needs information regarding which individual class changes are supported by the default state transformations of the DSU.

Additionally, the system can potentially be configured to postpone the updates from happening until the runtime phenomena are not possible: until the runtime state transformations are manageable by the DSU at hand.

This approach will use the runtime analysis to determine if the update might lead to the phenomena and apply only safe updates. Otherwise, the update is not continued and rescheduled after a short delay. When a window of opportunity appears when the runtime state changes and the state transformation capabilities of the DSU fit it better.

The exact details of such a setup require more research. Using the proposed system to analyze the practical data for applying development time DSU solutions will provide the necessary experimental data for the research of the production systems.

The ability to predict the runtime phenomena by observing the current runtime state of the application allows us to make the daunting task of developing DSU friendly applications easier. Moreover, the ability to identify updates that are complicated for the DSU solutions can be used to collect a corpus of update scenarios for a comprehensive DSU benchmark for Java programs. The existing research of the unified DSU benchmarks concentrates on the updates of the production systems written in lower level languages.

Dynamic Software Update is a complex problem, so any advancement in making it more widespread is a good step forward. Which makes the current work of enhancing the availability of DSU solutions that provide essential information about the predictability and applicability of an update significant.

7 RELATED WORK

Here we summarize prior work on dynamic software updates focusing on alternative DSU approaches or on the work emphasizing the safety of the updates to provide the context for this paper.

The field of analyzing the safety of the dynamic updates is not particularly new, but the work is mostly focused on safeguarding the updates to the production and mission critical systems. Such goal requires rigorous proof of the safety criterias and is not focusing on making the updating system easy and fast to use.

Gregersen et al. have identified which changes to the Java programs can lead to observable runtime phenomena (Gregersen and Jørgensen, 2011).

Bazzi et al. research the state mapping problem for DSU and try to limit that to make practical automatic solutions possible, (Bazzi et al., 2009).

Zhao et al. devise an automated static analysis system, (Zhao et al., 2014), that suggests safe points to developers, which can be run on the arbitrary applications without prior specification of the original safe points.

There also exist multiple dynamic software update solutions for the JVM programs, (Subramanian et al., 2009), (Gregersen and Jørgensen, 2009). For example, one of the newer ones, Rubah uses bytecode rewriting to enable dynamic update on the stock JVM, (Pina et al., 2014). The state is transformed either eagerly leveraging the parallelism of the JVM to achieve a speedup or lazily after the update. However, Rubah requires programmers help to make applications updatable, which makes it less attractive for the development time updates.

Then there is JRebel, a state-of-the-art DSU solu-

tion for JVM capableof reloading all changes to the application, (Kabanov and Vene, 2014). However, JRebel does not implement any non-trivial measures to ensure runtime phenomena free updates, which made JRebel a perfect candidate for a stock DSU solution to be enhanced with Genrih.

Rather than trying to devise a generic solution capable of updating the running application at any moment of time, another approach to sanitizing dynamic updates, is to allow developers to specify the "quiescence" safe points in the application code. For some application architectures, like the event-driven systems, such approach, is incredibly straightforward and does not require extensive changes to the application architecture. Hayden et al. integrated multithreaded quiescence into Kitsune and experimented with updating a real life event driven system to evaluate the performance of the approach. The results suggest that in an event-driven system it is relatively easy and fast to catch all the threads into a safe point, (Hayden et al., 2012b).

Another Hayden et al. work focuses on techniques for establishing the correctness of the dynamic updates, (Hayden et al., 2012a). They present a methodology for automatically verifying the correctness of dynamic updates using specifications provided by developers. The main approach lies in the provably correct merge transformation of the old and new versions of the code into a merged entity; that is later analyzed both statically and using a symbolic executor for the correctness of the dynamic update.

Giuffrida et al. introduced a system for live updates that uses time-travelling snapshot techniques to maintain the balance and transfer the runtime state back and forth between two versions of the code, (Giuffrida et al., 2013). Their approach also trades the time for an update for its safety, and not particularly applicable to the development time DSU.

The main focus of the existing research on DSU is concentrated either on making the production system updates more timely and safe by introducing manually or statically determined safe points, or encouraging to create the DSU aware applications by following a certain programming approach. However, to the best of our knowledge, inspecting runtime state of a statically typed object-oriented runtime to determine if an update is safe has not been previously discussed in details.

8 CONCLUSION

In this paper, we investigated an approach to the safety of the dynamic software update of the statically typed object-oriented programs. Previous research has determined that applying updates to the running code on the fly can result in the visible application-level side effects due to the runtime phenomena occurring after the update. The main reason for the phenomena to occur is breaking application assumptions about the runtime state because the update cannot automatically convert all the runtime state.

The main contribution of the current research work is the design and the implementation of a runtime analysis system that can automatically decide if the state transformation functions of the given DSU is sufficient for the current update. It is orthogonal to the actual solution performing the dynamic update and depends on just a handful of information about the update mechanics. The analysis system takes into account the exact changes an update consist of and the current runtime state to determine what runtime phenomena can be observed if the update is applied immediately.

We provided a construction of a mapping for the types of changes to a Java program that has been identified as capable of breaking the update by introducing an observable runtime phenomena. We implemented a prototype of the system that is capable of predicting the possibility of these phenomena occurring. The system prototype was integrated with JRebel dynamic updating functionality for the Java applications.

The main result of the system work is that it qualifies the DSU friendliness for the particular updates. At the same time, it provides preemptive feedback describing the runtime phenomena possibly caused by the update. Together these function can eliminate a major time waste the existing dynamic software update solutions for the development environment are susceptible to – debugging errors introduced by the runtime phenomena of the dynamic updates.

REFERENCES

Bazzi, R. A., Makris, K., Nayeri, P., and Shen, J. (2009). Dynamic software updates: the state mapping problem. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades (HotSWUp '09). ACM, , NY, USA, , Article 7 , 2 pages. DOI=*.

Erlang (2017). Erlang reloading documentation. http://www.erlang.org/doc/reference_manual/code_loading.html. Accessed: 2017-05-17.

Giuffrida, C., Iorgulescu, C., Kuijsten, A., and Tanenbaum, A. S. (2013). Back to the future: fault-tolerant live update with time-traveling state transfer. In *Proceedings of the 27th USENIX conference on Large Installation System Administration (LISA'13)*, pages 89–104, Berkeley, CA, USA. USENIX Association.

Gregersen, A. R. (2011). Implications of modular systems on dynamic updating. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering (CBSE '11). ACM, , NY, USA,. DOI=10.1145/2000229.2000254*, pages 169–178.

Gregersen, A. R. and Jørgensen, B. N. (2009). Dynamic update of java applications - balancing change flexibility vs. programming transparency. *J. Softw. Maint. Evol.*, 21(2):81–112.

Gregersen, A. R. and Jørgensen, B. N. (2011). Run-time phenomena in dynamic software updating: causes and effects. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL '11). ACM, , NY, USA,. DOI=10.1145/2024445.2024448*, pages 6–15.

Gupta, D., Jalote, P., and Barua, G. (1996). A formal framework for on-line software version change. *IEEE Trans. Softw. Eng*, 22(2):120–131.

Hayden, C. M. (2012). Clear, correct, and efficient dynamic software updates. *Ph*, 3543.

Hayden, C. M., Magill, S., Hicks, M., Foster, N., and Foster, J. S. (2012a). Specifying and verifying the correctness of dynamic software updates. In tools, e. V. S. T. T. E. ., Joshi, R., Müller, P., and Podelski, A., editors, *Proceedings of the 4th international conference on Verified Software: theories*, pages 278–293. Springer-Verlag, Berlin, Heidelberg,. DOI=10.1007/978-3-642-27705-4_22.

Hayden, C. M., Saur, K., Hicks, M., and Foster, J. S. (2012b). A study of dynamic software update quiescence for multithreaded programs. In *12). IEEE Press, Piscataway, NJ, USA*, pages 6–10. Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp.

Kabanov, J. and Vene, V. (2014). A thousand years of productivity: the jrebel story. *Softw: Pract. Exper.*, 44:105–127.

Oleg Šelajev, R. R. and Kabanov, J. (2013). Static analysis for dynamic updates. ACM, New York, NY, USA. Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '13).

Oracle (2017). Jvmti documentation. http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html#whatIs. Accessed: 2017-05-17.

Pina, L., Veiga, L., and Hicks, M. (2014). Rubah: Dsu for java on a stock jvm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14). ACM, , NY, USA,. DOI=10.1145/2660193.2660220*, pages 103–119.

Shipilev, A. (2017). Java microbenchmark harness. http://openjdk.java.net/projects/code-tools/jmh/. Accessed: 2017-05-17.

Subramanian, S., Hicks, M., and McKinley, K. S. (2009). Dynamic software updates: a vm-centric approach. *SIGPLAN Not*, 44(6):1–12.

Würthinger, T. (2010). Wimmer. In *10). ACM, New York, NY, USA,. DOI=*, pages 10–19, L Dynamic code evolution for Java. In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ. C. and Stadler.

Zhao, Z., Ma, X., Xu, C., and Yang, W. (2014). Automated recommendation of dynamic software update points: an exploratory study. In *Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware (INTERNETWARE 2014). ACM, , NY, USA,. DOI=10.1145/2677832.2677853*, pages 136–144.