Towards Change Validation in Dynamic System Updating Frameworks

Mehdi Jalili Kordkandi

Graduate School in Computer Science
PhD in Computer Science

PhD School Headmaster: Prof. Paolo Boldi

Advisor: Prof. Walter Cazzola

UNIVERSITÀ DEGLI STUDI DI MILANO

Department of Computer Science "Giovanni Degli Antoni"

ADAPT-Lab

Cycle XXIX
INF/01 Informatica
Academic Year 2016–2017

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

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. Determining automatically a safe moment, the time that the updating process could be started, is still an open crux that usually neglected from the existing DSU systems. The program developer is the best one who knows the program semantics and the logical relations between two successive versions as well as the constraints which should be respected in order to proceed with the update. Therefore, a set of meta-data has been introduced that could be exploited to explain the constraints of the update. These constraints should be considered at the dynamic update time. Thus, a runtime validator has been designed and implemented to verify these constraints before starting the update process. The validator is independent of existing DSU systems and can be plugged into DSUs as a pre-update component. An architecture for validation has been proposed that includes the DSU, the running program, the validator, and their communications.

Along with the ability to describe the restrictions by using meta-data, a method has been presented to extract some constraints automatically. The gradual transition from the old version to the new version requires that the running application frequently switches between executing old and new code for a transient period. Although this swinging execution phenomenon is inevitable, its beginning can be selected. Considering this issue, an automatic method has been proposed to determine which part of the code is unsafe to participate in the swinging execution. The method has been implemented as a static analyzer which can annotate the unsafe part of the code as constraints. This approach is demonstrated in the evolution of the various versions of three different long-running software systems and compared to other approaches.

Although the approach has been evaluated by evolving various programs, the impact of different changes in the dynamic update is not entirely clear. In addition, the study of the effect of these changes can identify code smells on the program, regarding the dynamic update issue. For the first time, the code smells have been introduced that may cause a run-time or syntax error on the dynamic update process. A set of candidate error-prone patterns has been developed based on programming language features and possible changes for each item. This set of 75 patterns is inspected by three distinct DSUs to identify problematic cases as code smells. Additionally, error-prone patterns set can be exploited as a reference set by other DSUs to measure own flexibility.

Acknowledgement

It's my pleasure to acknowledge those people who have; mostly out of kindness, supported me during my Ph.D. I'm really indebted too much for their encouragement and support.

My deepest gratitude is to my supervisor, Prof. Walter Cazzola for his valuable advice, supports, and encouragement during my P.h.D. His patience and support helped me overcome the critical situations and finish this dissertation.

I would like to really appreciate the reviewers of my thesis, Prof. Gunter Saake, Otto-von-Guericke University; Prof. Paola Giannini, University of Turin; and Prof. Hooman Tahayori, University of Shiraz; for their thorough reviews and suggestions, which contributed to improving the quality of my work. I truly appreciate the time they spent and their efforts.

Furthermore, I would like to thank my family for all their love and encouragement. Especially my parents who raised me with a love of science and supported me in all my pursuits and also my mother-in-law, brothers, and sister. Most of all I should really appreciate my loving, supportive, encouraging, and patient wife Maryam and my little king, Ilya who have supported me faithfully during the final stages of this Ph.D. Thank you.

I gratefully acknowledge the scholarship received towards my Ph.D. from Università Degli Studi di Milano.

Contents

List of Figures

List of Tables

1 Introduction

Maintenance is one of the undeniable and expensive phases of software developing process. The operation and maintenance phases totally include 67% of a program life-cycle cost [15]. Programs need to be evolved in order to fix the bugs or to add new functionality because of changing user requirements, improving performance, etc. Nowadays by the ubiquitous growing of software systems, as well as increased online services, motivating the quest for faster updating. Security bugs need to be fixed A.S.A.P to prevent the penetration of malicious entities.

The typical way to update a program is stopping the running program, modifying the code and then restarting the updated version (*cold restart* [56]). This approach is not always acceptable. Stopping the execution of some kind of programs could cause pecuniary losses or life-threatening risks. Online transaction systems, life-support systems and so on are placed in this category. *Dynamic Software Updating (DSU)* [67] addresses this issue by changing a program at run-time without stopping its execution. DSU has appeared with other phrases such as online version change [17], on-the-fly program modification, hot-swapping [36] in the literature.

Applying the DSU approach is not only limited to evolve the highly available applications in the deployment phase; this approach even can be employed in developing phase of a software system. When a developer performs a small change on an under-developing application, the program should be stopped and run again to observe the modification impacts. This process could be time-consuming, especially in the case of complicated programs. In this application, DSU can be applied for *edit and continue* purpose. The developed DSU tools for this approach in most of the cases is tightly integrated with IDEs. For instance, JRebel [72] and JavAdaptor [106] have been developed to support DSU in Java language and provide plug-ins for IDEs such as Eclipse and NetBeans.

1.1 Dynamic Software Updating

Over the past two decades, many efforts have been carried out to develop DSU mechanisms in various aspects of software development. However, programming languages have been more prominent. These systems provide a solution to update programs written in a specific programming language dynamically. Some of the programming languages are *dynamically typed* and support this issue intrinsically such as Erlang[12] and Smalltalk[48]. Also, some other languages such as UpgradeJ[23] have been designed to support the DSU explicitly. The issue of applying this approach is that the program should be written in these languages that usually might not be so popular.

Providing DSU solutions for General Purpose Languages (GPLs) is highly regarded because a lot of long-running applications have been developed in this kind of languages. One of the most important GPLs is C programming language. This imperative language is embraced by the programmer because it gives more flexibility on performance and memory management. Several systems have been proposed to support non-stop updating in C language programs without changing the syntax and semantics of the language [11, 116, 69, 87, 99]. Furthermore, the kernel of some operating systems are developed in C and some DSU techniques have been presented for operating systems to provide this capability to apply patches without restarting [31, 19, 47, 13].

Java is another important general-purpose programming language that has drawn the most attention from the programmers in the past 15 years [9]. Many long-running applications have been developed in Java that continuous servicing is a critical requirement. Considering the issue that this object-oriented programming language does not support dynamic update intrinsically, on-the-fly update techniques for Java programs are demanded. Java code usually is compiled into bytecode, which can be executed by a Java Virtual Machine (JVM). So, DSU systems for Java language have been developed on two levels: JVM level and bytecode level. JVM level solutions can be implemented by customizing Garbage Collector and Just-In-Time(JIT) components. These solutions tightly depend on a particular JVM and this can be a disadvantage. Tools like HotSwap [37], JVolve [117], DCE VM [126] and Rubah [101] are implemented at the JVM level. The bytecode level solutions include techniques that provide the dynamic update for Java programs without modifying JVM. In these solutions, the DSU usually rewrites the program code and adds a level of indirection by exploiting some techniques such as proxies and containers. The advantage of these methods is preserving the portability of programs, and the disadvantage is a probable reduction in performance due to code manipulation.

JavAdaptor [106], DUSC [97] and JRebel [72] are placed in this category.

Although the main purpose of these systems is avoiding the program from stopping during the upgrade, some other issues are prominent in this context such as flexibility, type safety, update point, state transformation, and so on. In the flexibility context, various kind of changes may occur in a program modification. However, not necessarily all of the changes are supported by some DSUs. For instance, HotSwap only accepts the change in a method body whereas JRebel supports most of the changes. Another relevant issue that can be taken into account is type safety. In the statically typed language such as Java, each item's type should be determined at the compile time and it should keep the type during its lifetime. The proposed methods for the dynamic update should not violate this rule. Update point is also one of the common issues in most DSU systems. When the new version of the program is ready, the DSU system should decide on the start time of performing the update. It can be started immediately, either at a predefined point or with respecting to specific constraints. In perspective of state transformation, DSUs should provide a mechanism to transform values from old instances to the new ones. It can be performed in automated or assisted ways.

In addition to above issues, one of the most important concerns of DSU systems is how to make sure that the dynamic evolution process is done without introducing an error. This error may stop the update process and the running program is crashed by creating a runtime error. This is a disaster for the long-running programs that the high availability is a critical property for them. In a volatile mode, the running program may confront with a transient inconsistency and expose some wrong behaviors. Even this mode might not be acceptable on the most systems and causes semantic errors. Ensuring that the update process is performed without any runtime or semantic error is called *validation* or *correctness*. Gupta et al. introduce the notion of update validity [59]. They prove that generally, it is undecidable to determine if a given arbitrary update is valid or not. They define the validation issue based on the ability to reach some states of the running new program (the new version of the program started from the initial state) by the dynamically updated program. 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*). Gupta et al. introduced the *reachability* property to define an equivalence between these two approaches. 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. Most DSU systems have not considered validation issue and so far DSU solutions have not been very practical. In this work, some steps towards a valid dynamic update have been taken.

1.2 Validation Issue

Although 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. Each 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 one who knows the program semantics and the logical relations between two successive versions as well as the constraints which should be respected in order to proceed with the update. Turning one version into another is safe only when the changes are to be exerted with respect to the imposed constraints. Therefore, for each program, a dedicated collection of constraints should be introduced 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 keep the program stable.

One of the constraints that can be identified is to specify unsafe points to start the update process. This means that while the program is executing a specific part of the code, starting the update process will lead to a fatal error and the dynamic update process fails. Let us explain with an example how choosing a wrong time to start the update may cause the running application is crashed and why some parts of the code are unsafe to start the update. Suppose in a program, `foo` method calls `bar` method without parameter. In the new version of the code, `bar` method is changed and a parameter is added to its signature. Also `foo` method body is adapted to call `bar` method with an appropriate argument. The old and new versions of this code can be executed individually without any error. Let us consider the old version of the application is executing `foo` method and exactly before calling `bar` method the new code is replaced dynamically. While the new versions of both methods have been replaced, the program continues to execute the old `foo` method that is on the call stack. The codes in the call stack cannot be updated. So, the program will attempt to call the old version of `bar` method which does not exist in the memory and the program terminates abnormally with `NoSuchMethodException`. To avoid this situation, the update should be postponed until the execution of the `foo`

method is completed. In this example, the `foo` method is unsafe to start the update.

Therefore, some facilities should be provided for developers to express these constraints. The easiest and most convenient way is to introduce these meta-data within the application code since their evaluation is a part of the application execution and its updating. In Java, this means to use Java annotation facility. So, we introduce a set of annotations that could be exploited to explain the constraints such as determining unsafe update points. These meta-data can be easily used to express both static or dynamic constraints. Although static constraints include some conditions that are specified before starting the deployment process, dynamic constraints depend on the status of the running application.

According to the semantic relations between two versions of a program, the developer can decorate the program code with the provided annotations. These constraints should be considered at the dynamic update time. We design and implement a runtime validator to verify these constraints before starting the update process. Regarding all the specified restrictions, we can reach a safe update point to start deployment. The validator is independent of existing DSU systems and can be plugged into DSUs as a pre-update component. We propose an architecture including the DSU, the running application, the validator, and their communications. Moreover, we will explain how each annotation should be processed as well as the main algorithm for finding a safe update point.

Along with the ability to describe the restrictions by using annotations, it should be noted that the annotating process 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. 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. Therefore, we study the execution model of the program in the dynamic update process to find a way to automatically annotating.

When a new version of an application is ready, DSU tool starts the updating process. While the application is running the old code, the DSU system deploys the new version of the 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. So that, the application switches between the execution of the old and the new code consistently. We call this phenomenon *swinging execution*. This situation continues until all of the call frames in the stack pointers refer to the new code. During swinging execution between old and new code, the application execution can manifest some flaws and its state is potentially inconsistent. As shown earlier in an example, this transient inconsistency may lead to a fatal error and update failure. However, some DSUs [106, 51] ignore this phenomenon and accept such a risk which may not always be acceptable in the critical application. On the other side, some DSU systems [97, 117] 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 from that postponing the update is not always feasible or desirable, it may lead to infinite waiting.

Theoretically, each part of the program code can participate in the swinging execution. In fact, at each step of this phenomenon, a part of the old code (on the call stack) along with the new code is affected by the running program. However, some code participation may introduce a runtime error at the update time. Considering this issue, an automatic way has been proposed to determine which part of the code is unsafe to start the update process and participate in the swinging execution. This novel approach statically anticipates swinging execution impact on each changed part of the code and determines unsafe codes. This method is implemented as a static analyzer that takes two versions source code and gives an annotated code.

In order to demonstrate the presented approach, we considered various versions of three different long-running programs and their dynamic evolution to the next version. 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. We compare our approach with immediate and conservative approaches. In addition, possible update errors are classified and the causes of their occurrence are explained. This part of work is published in the following publication [28]:

– Walter Cazzola and Mehdi Jalili, "Dodging Unsafe Update Points in Java Dynamic Updating Systems", *in Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE'16),* Alexander Romanovsky and Elena Troubitsyna, Eds., Ottawa, Canada, October 2016, IEEE.

Although we evaluate our approach in different versions of the three server programs, the impact of various changes in the dynamic update is not entirely clear. Some types of modifications in the program cannot have a negative effect

on its dynamic updating. Moreover, the study of the effect of these changes can identify code smells on the program, regarding the dynamic update issue. The term of *code smell or bad smell* usually refers to any symptom in the program source code that probably portends a deeper problem[121]. Code smells usually are not a bug and they do not interfere the normal execution of a program. However, they may cause other problems such as performance penalty or increase the risk of bugs or failure in the future. Smells can be considered in the perspective of creating a probable error in the dynamic update process, where some specific changes in the program may cause a fatal error at the update time. For the first time, we introduce the code smells that may cause a run-time or syntax error on the dynamic update process.

To achieve this goal, we first studied the Java language features and the changes that can be applied to each item. Then, we designed and developed a set of error-prone patterns. Each pattern nominates an atomic simple change that can be occurred in a program evolution. We develop more than 75 individual error-prone patterns based on Java language features and possible changes for each item. This collection is divided into three categories: i) changes that cannot be applied to some items. For instance, changing a value is only possible in a field of the class and it is not possible for the methods of a class. ii) the changes that can be applied to an item that cannot participate in the swinging execution. For instance, when a method is added to a class, it is not accessible from the old code. Thus, it does not involve in this context. iii) the changes that might be involved in the swinging execution. The error-prone patterns are the potential candidates to be recognized as a DSU code smell. In addition, we believe that these patterns can be used by other DSU tools to measure their flexibility.

We then executed these candidate error-prone patterns on three different DSU tools. At the beginning, the patterns were filtered according to the acceptance criteria of the DSUs, since all of the changed are not supported by the DSUs. Then the probable run-time or syntax errors were considered. Runtime errors cause the program to crash, but syntax errors violate programming language rules. For instance, an illegal access might be occurred after reducing the visibility of an element. It does not cause a runtime error, but it is a semantic error due to a violation of the privacy of the element. However, the problematic patterns can be identified as code smells. In the next step, these detected patterns are investigated by our proposed method to determine the status of error detection in the patterns. Finally, we enhanced our static analyzer in order to detect all of the code smells. This part of work is ready to submit as a journal paper.

1.3 Contributions

The major contributions of the dissertation are as follows:

1. Proposing a set of meta-data that can be exploited by the developer to express the constraints. The DSU should be in charge of verifying these constraints before the updating.
2. Introducing a runtime validator to verify the predefined constraints at the update time. Respecting to all of the specified restrictions, we can reach a safe update point to start deployment.
3. Determining the criteria for identifying the unsafe parts of the code by studying the swinging execution phenomenon. This phenomenon makes a transient inconsistency at the update time.
4. Proposing an automatic method to determine unsafe update points by processing source code through an implemented static analyzer.
5. Studying each atomic change that can happen in a program evolution based on Java language features and possible changes for each item. We develop more than 75 candidate error-prone patterns. The pattern set is used to explore code smells on dynamic software updating. Moreover, it can be exploited as a reference set by other DSU tools.
6. Extracting the code smells of DSU process, based on run-time and syntax errors which may happen at the update time due to the swinging execution.
7. Enhancing proposed static analyzer to detect all of the code smells.

1.4 Thesis Structure

In this chapter, the dynamic software updating was briefly described along with the problem of validation the DSU systems. In the following chapters, we will explain the details of our approach. The chapters are structured as follows:

In **Chapter 2**, background information in the field of research has been provided. It includes the explanation of dynamic software updating in general and its challenges. Some of the mechanisms used by DSUs to provide dynamic evolution are also explained. The issue of validation is illustrated through an example.

In **Chapter 3**, first of all, we introduce a set of meta-data that could be exploited to explain the update constraints. Then we present a validation architecture that includes a validator to verify defined constraints. This validator finds a safe update point at run-time thanks to the provided constraints. Finally, an automatic method is proposed to determine the unsafe update

points through a static analysis. The proposed method is applied to different long-running programs and the results of the experiments are discussed.

In **Chapter 4**, regarding the dynamic update issue, we determine code smells on the program. These smells may cause a run-time or syntax error on the dynamic update process. To achieve this goal firstly, we write a set of error-prone patterns. Then, we run these patterns on three DSUs and identify problematic patterns. Finally, we enhance the static analyzer to cover all of the code smells.

In **Chapter 5**, various DSU systems in Java are described briefly, and the policy of each system is explained in the face of validation issue. Moreover, we discuss the state-of-the-art of validation problem.

Finally, in **Chapter 6**, we briefly explain the results and proposes some future works.

2
Dynamic Software Updating

In this chapter, we introduce the dynamic software updating problem, its challenges, and validation approach in the dynamic update.

2.1 Definition

Software evolution is defined as all programming activities which are intended to produce a new software version from an earlier operational version [81]. After deploying the first release of a system, the program needs to be evolved in order to fix bugs, respond to the new requirements and improve the performance. With the prominent role of software in various aspects of life, rapid changes in the user needs, the occurrence of security holes, system interconnections, etc., in many cases, update becomes a daily activity. The cost of software evolution activities can range from 50% [82] to 90% [39] of the total development cost that a part of this cost is related to the reinstallation process.

By increasing diffusion of online services, high availability has become one of the most needed features for software systems. Typical updating process interrupts the service provision to permit the switch between the current and new version of the software system (*cold restart* [56]). This is a common practice even if often undesired. No matter how short and easy the updating process is, such a way to proceed has two major drawbacks: i) the application system is unavailable during the update and ii) its state is lost or must be migrated. Such drawbacks are unacceptable in many highly available applications where they could either cause financial losses [112, 95], increasing maintenance costs [129], endangering people and things or at least causes dissatisfaction of the users. Telecommunication switches, financial transaction system, airport traffic control systems, oil and gas production, power generation, smart-grids and so on are in this category.

Another consideration comes in the process of developing a software. Developers usually use the IDEs to write the programs. For each small change, they need to recompile the program, relaunch the application and find out how

the changes have affected the program. This process may be repeated every several minutes during developing an application. This process wastes the developer time. In the enterprise large application, this time is more significant. Programmers spend an average of 5.77 minutes per hour on redeployment process[73]. This constitutes 9.6% of the coding time.

Dynamic Software Updating (DSU) includes some techniques that permit a program to be adapted during its execution without stopping it [67]. Fabry was the first one who notice this issue by introducing an *on the fly module changes* system for abstract data types written in the procedural languages [40]. Later Kramer and Magee proposed a model for dynamic change management which isolates structural concerns from component application concerns [79]. It is conceivable that there is no standard term to express these techniques. Various terms have been used to name these systems in different references. Gupta used *on-Line Software Version Change* phrase in his work for updating object oriented systems dynamically [59]. Other synonyms which are used in the literature include: *Runtime evolution* [35, 61, 36], *Dynamic (software) evolution* [96, 38, 126], *Runtime adaptation* [57, 90] and *Dynamic deployment* [109]. However, most frequently used term is *Dynamic (Software) Updating* [102, 97, 67, 115, 65, 27].

2.2 Different Aspects

The problem of dynamic software updating can be considered from the different point-of-view that follows.

2.2.1 Intrinsic Support by Programming Language

The ideal mode for providing DSU facility is the programming languages level. In this case, the programmers develop the applications without any concern about dynamic update problems and take advantage of the direct development of highly available systems without any extra effort in development. This is a major advantage and allows for flexible modification in a running program. These dynamically-typed languages usually provide facility to specify the point to switch to the new version. Moreover, the developer can develop a lazy update semantic which causes the minimum pause in the execution of the running program.

Smalltalk [48, 49] and **CLOS** [62, 43] are two examples of this kind of languages. They permit a class to be redefined in a dynamic-typed system. Meta-classes in these languages describe the behavior of other classes. Smalltalk

provides a *meta-object protocol (MOP)* that can describe any aspect of language. MOP supports DSUs by *Object* and *Behaviour* classes. The dictionary of class's methods is kept in class *Behaviour* while the program is executing. The dictionary can be searched and modified at runtime. Moreover, Smalltalk allows the programmer to inspect the program call stack and modify it. CLOS redefines a class by defining a new class with the same name. After redefining a class, the slots of the class may change, then CLOS disseminates the changes to the instances and subclasses of the modified class.

Erlang [12] is another programming language that support DSU natively. Erlang code is constructed with modules and each module is described as a set of functions. Erlang supports dynamic update by replacing code at runtime at the module level. The code replacement mechanism is made on the top of dynamic module loading. The new modules are loaded dynamically while the program is running. Each module in Erlang can have two versions: the current and old version. The newly loaded module is current. When a new version of the module is loaded, the code of the previous one becomes old and the new code turns into the current.

UpgradeJ [23] is an extension to the Java programming language which allows classes to be updated dynamically. This language level solution for DSU annotates every class with its version. Instances can be declared as *exact* or *upgradable* in a written program. An exact instance always exploits the same version of its class whereas upgradable instance uses the latest version of its class.

Although direct language support of DSU is a major advantage, these programming languages are not so common. It can be a major disadvantage of this issue. Moreover, languages like Python [21] and Ruby [120] that support DSU natively are so slow in comparison with general-purpose language like Java. Python and Ruby almost 41 times slower than Java [42].

2.2.2 Formal Approaches

Several works have been proposed to define a formal model of DSU process and inspect its features. Gupta et al. [60] was the first one who proposed a framework to model online program changes and proved some properties. Gupta defines the dynamic update as a following:

Definition. An online change from program Π to Π' at time t using the state mapping S, in process P (executing Π) is equivalent to the following sequence of steps:

1. P is stopped at time t in state s.

2. The code of P is replaced by the program Π', its state is mapped by S and then P is continued from state S(s) and with code Π'.

Gupta explains the correctness of DSU process by defining the *reachability*. According to this definition, the updated program finally should reach the same state that new program could reach. The author proves that this issue is undecidable in general case. In another work, the correctness of update is defined as preserving old behaviors by the updated program [79]. However, this description is insubstantial because some behaviors of the program are changed/removed due to the fixing bugs and adding new functions [24].

Bearman et al. [22] introduce a small update calculus with an accurate mathematical semantic. This is an extension of the first-order simply-typed lambda calculus with mutually-recursive modules and a primitive for updating them. It allows modules to be updated in the system, including changes to the types and their definitions, as long as the resulting program remains type-correct. The programmer can control the update time by inserting an update primitive as well as control the update effects using the proper variable syntax.

Another calculus is Proteus [116]. It is employed to model type-safe dynamic updates in procedural languages. Proteus supports dynamic update on functions, named types and data. It exploits the concept of *con-t-freeness* to investigate the type safety of an update in the named types changes. A certain update point is con-t-free if after the update, for every type t, if the program will never use concretely the old value of type t. It means that non-updated code will not use t concretely beyond the update point and thus t's representation can be changed safely. This analysis is implemented for C programming language. Later an extension of Proteus is proposed to support multi-thread programs. Proteus-tx [92] introduces a new correctness property named *transactional version consistency* (TVC). In this approach, the developer can specify some blocks of code as transactions that should always execute in one version of the program. Therefore, an update can take place within a transaction if the transaction's execution is continued either in the old or new version of the program.

Boyapati et al. [25] propose an automatic way to upgrade objects on persistent object stores [16]. For performing an upgrade in these systems, the programmer usually defines a transform function for each class whose objects need to be upgraded. This work illustrates some upgrade modularity conditions that impose the behavior of an upgrade system. These conditions should be satisfied in any upgrade system. It guarantees that in the case of running transform function, it only meets object interfaces and invariants that their upgrades were defined. They describe a prototype implementation that supports fully

expressive, modular, and lazy upgrades.

2.2.3 Procedural Languages

DYMOS [80] is a programming system that allows StarMod programs to be modified dynamically. StarMod language is an extension of Modula [123]. To update procedures dynamically, procedures should be modified and recompiled by the programmer and then the system changes the current core image to the new code and data. It contains a command interpreter that can apply update process based on determined conditions, e.g., when certain procedures are inactive.

Frieder and Segal implemented procedure-oriented dynamic updating system (PODUS) [113]. Updating a program in PODUS includes two steps: first, loading the new version of the program; second, replacing old procedures with their corresponding new procedures during execution. The program update process is completed if all procedures are replaced by their new versions. PODUS prevents to start an update process if one of the updated procedures is executing.

The problem of DSU is particularly important for GPL because long-standing applications are implemented in this kind of languages. One of the most important languages is C. Various systems have been developed to support dynamic updating in C programs. OPUS [11] is a tool for applying software patches to a C program at runtime. In fact, this system is customized for dynamically applying the security patches to interactive applications which are the potential targets of security threats. By putting a limitation on the type of patches accepted by this system, they diminish extra programmer tasks which normally should be done in the developing and testing of a conventional stop-and-restart patch.

Neamtiu et al. developed a dynamic updating for C programs called Ginseng [93, 91]. It is very flexible and supports lazy state migration. Ginseng uses function indirection and type-wrapping techniques to make a program updateable. The program is compiled by a special compiler and the programmer should insert the update points inside the source code before compile. These modifications impose performance penalties up to 32% at the runtime compared to the non-updatable version of the program. Ginseng has some limitations in increasing the size of existing structure. When the size of an updated structure reaches the maximum size, the program should be restarted to accept future increases. Moreover, it imposes some programming style limitations to satisfy its static analysis.

Another system is POLUS [32], a POwerful Live Updating System for C-like programs. It supports dynamic update for existing binaries and already running applications. POLUS uses a free consistency model to allow the active changed code to be updated. Active functions continue their execution at the old version. It may lead to type safety violation and hinder to update long-running loops. State transformation functions should be written by the programmer to ensure system consistency.

Mariks et al. introduced UpStare to replace active execution with the new version entirely by reconstructing the stack [87, 86]. It needs the developer to determine the corresponding points between the old and new versions of the program as well as write transform functions to convert the stack of all active functions into the new version. It offers a flexible update because the new method can be started from the different point of old method. UpStare burdens up to 38.5% overhead due to the extra indirections and update point checking. Moreover, it is impossible to switch to the old version and developer is responsible for checking the semantic correctness of function mapping.

Kitsune [66] is another system to support DSU for C programs. It is very flexible and permits any modification in a program. The programmer should determine some points inside the code where the update can take place. Kitsune checks the probable update at these points. When an update is available, all the running threads of the program will be stopped at the update point. Kitsune traverses the heap and runs user-specified transform code to migrate program state. Finally, all the threads restart to run with the new code. Specifying update points and writing transform functions by the developer without any validation mechanism makes the update process prone to error. In addition, immediate program state transformation at the update time increases the pause time of the running program which may be significant and unacceptable in the large programs.

Replus [30] is a DSU framework that has been developed recently, balances practically and functionality. Like Kitsune, this system manipulates the stack of program's thread. It uses two mechanisms: first, immediate stack updating that updates stack of a thread instantly. Second, timely stack updating that only update the stack frames of essential functions without affecting others. Moreover, Replus introduces an instruction level updating mechanism to apply security patches impressively.

2.2.4 Operating Systems

Operating Systems (OS)s play the core role in software systems. Operating system vendors release patches frequently to reduce the vulnerability of end-user systems. However, the common process for applying the patches is downloading the online patch, installing it, and restarting the system to take effect of installing new patches. Increasing the frequency of releasing new patches makes applying process a burdensome for patch recipients. Particularly, restarting the operating system at the bottom of the system stack causes losing the system state. Many efforts have been made to apply new patch on-the-fly without restarting the OS.

Baumann et al. [20, 18] implemented a dynamic update mechanism in the K42 research operating system developed at IBM. K42 is almost entirely object-oriented operating system supporting hot-swapping and written in C++. The proposed system permits the update of both the kernel code and the data structures. It performs the dynamic update at the class level. Each class that might be updated should include the state import and export methods. Upon update process, the new version of class imports the exported old state. The import and export methods are written manually.

LUCOS [31] is a version of POLUS that uses Xen-based virtualization techniques to provide a dynamic update on Linux. Extracting the two version changes and constructing the update are done manually. All the references to the old functions are modified and redirected to the new versions by stack-walking and binary rewriting techniques. After the update, on accessing an old type value, a transform function is run to convert the old type value to the new type value. However, this tool suffers from manual patch construction as well as type safety violation due to the disregarding the updated functions on the stack.

DynAMOS [88] enables essential dynamic and adaptive software updates in a commodity operating system kernel without kernel recompilation or reboot. It exploits dynamic code instrumentation technique named *adaptive function cloning*. It prohibits active functions updates but permits the simultaneous execution of many versions of functions. However, reaching a safe update point may lead to an infinite waiting. It allows active data structures on the stack to be updated through *shadow data structure* but it needs data access indirection and preservation of semantics of the data. The approach is illustrated by dynamically updating core subsystems of the Linux kernel.

Ksplice [13] provides a dynamic update system for updating Linux kernel. It only supports function modification without changing signature by loading new functions as modules and redirecting the callers to the new version of functions.

It forbids active function update by scrolling threads stack. In addition, it compares kernel binary image files and generates DSU patches automatically.

2.3 Dynamic Update in Java

Java has been a very popular programming languages for the past 15 years [9]. Many applications developed in Java have continuing service as a vital requirement. Nevertheless, this object-oriented language does not support the dynamic update intrinsically.

Java applications usually are compiled to bytecode that can be run by a *Java Virtual Machine* (JVM) [83] regardless of computer architecture. So, Java applications benefit from "*write once, run anywhere*" (WORA) [10]. JVM provides a runtime environment for a Java program and supports memory management automatically through a *Garbage Collector* (GC). GC frees up the occupied memory that is no longer referenced by any reachable Java object. In addition, JVM exploits a *just-in-time* (JIT) compiler that compiles bytecode to a machine code which can be executed directly on the hardware. The DSU systems in Java have been introduced in two levels: i) At the code level by rewriting the program bytecode ii) At the JVM level by modifying the virtual machine to support DSU. We explain these two approaches with more details.

2.3.1 JVM Modification

The first feature to add for on-the-fly deployment of Java application is a modification of the Java Virtual Machine. Implementing a DSU system on JVM has the advantage that each program benefits directly from DSU privileges without the demand for any code manipulation. Usually, code modification decreases the performance of the program at runtime. Moreover, the reflection APIs can be used. On the contrary, the major disadvantage of implementing DSU system by JVM modification is portability violation. DSU systems commonly implement their solution in a particular JVM. It is quite hard and impractical to apply the developed patch to different JVMs automatically. Furthermore, these systems need to be maintained in the case of JVM upgrading. Especially if the JVM upgrade involves changes on the GC or JIT compiler sections. The system such as JDrums [110], DVM [89], HotSwap [37], JVolve [117], and the DCE VM [126] are implemented at the JVM level.

The systems that modify the JVM to support DSU on Java application benefit of two facilities: i) customizing the GC component to transform running program state. ii) customizing the JIT compiler to replace the outdated code.

Garbage collector of JVMs usually follows the transitive closure of reachable instances to specify which one can be reclaimed. Moreover, it copies objects between different generations according to their durability [70]. GC cycle provides the best opportunity to DSU system to find all the outdated instances and transform their states. This approach increases the size of outdated objects without the need for type wrapping or shadow structures techniques which are exploited at the bytecode rewriting level. DVM, JVolve, and the DCE VM customize GC to transform the state between old and new objects. However, instead of customizing the GC, JDrums enforces the JVM to keep an entry for each instance in an object table. It adds an extra level of indirection for accessing the objects. JDrums can traverse the object table to find intended object entry.

Another facility that can be employed by DSU system is JIT compiler. Ordinarily, during a program execution, a method of a program can be recompiled by the JIT compiler to optimize it. This process might be repeated several times over the program lifetime. Obviously, this dynamically recompiling process can detect calls to the outdated methods and replace them with the updated version. This is what dynamic update systems need. JIT compiler can be extended to support dynamic update by replacing the old methods with their new versions. In addition, some JIT compilers support On-Stack Replacement (OSR) by modifying return addresses and program counters. DSU system can ask JIT compiler to traverse the stack frames and adjust the return addresses of outdated methods with their new versions. HotSwap, the DCE VM, and JVolve customized the JIT compiler to support DSU. Whereas, DVM runs in the interpreted mode disabling JIT compiler.

2.3.2 Bytecode Rewriting

The second feature to provide DSU capability for Java programs is to rewrite the bytecode. This rewriting should keep the semantics of the original program. These systems exploit different mechanisms to redecorate the program code to overcome some language role violation problems that might happen on deploying the new version. They borrow the decorator and proxy patterns as a design pattern[46] in the object-oriented programming to create DSU facility on Java programs [107, 103]. For instance, DUSC [97], JavAdaptor [105], and JRebel [72] support DSU through bytecode rewriting.

Iguana/J [109] ports the concept of meta-class in Smalltalk and CLOS to the Java code and permits the programmer to define similar meta-classes in Java programs. The proposed architecture is flexible and supports unanticipated

dynamic modification. However, the authors admit significant slowdown in creating an object, calling methods, and returning methods.

A part of DSU systems exploit the proxy objects to implement class redefinition [97, 107, 52]. In general, the language level approaches have the advantage that the runtime environment does not need to be modified. Therefore, the dynamic deployment can be performed on any JVM that might run on different platforms and operating systems. The user can exploit JVMTI command to redefine the updated class. The main disadvantages are i) The significant performance penalty is introduced by the usual indirection that happens due to techniques are used. ii) Less flexibility on the type of changes, e.g. the class hierarchy change is prohibited on some systems[97]. iii) It is quite hard to trigger code evolution in the development environments or needs special plug-ins. iv) A probable reflection inside the code is not affected by the rewriting process and prone to make an error at runtime.

Besides proxy objects, another technique called the object wrapping is used. Each class should implement an interface that includes its public members. The class can be redefined but the corresponding interface should remain unmodified. The fields and variables that keep the instance object of the class must be defined as a corresponding interface type. Here also, additional indirection imposes a performance penalty on the system at runtime compared to the original application. Like the previous technique, Java reflection is not affected by this approach and it imposes some programming style restrictions to the programmer.

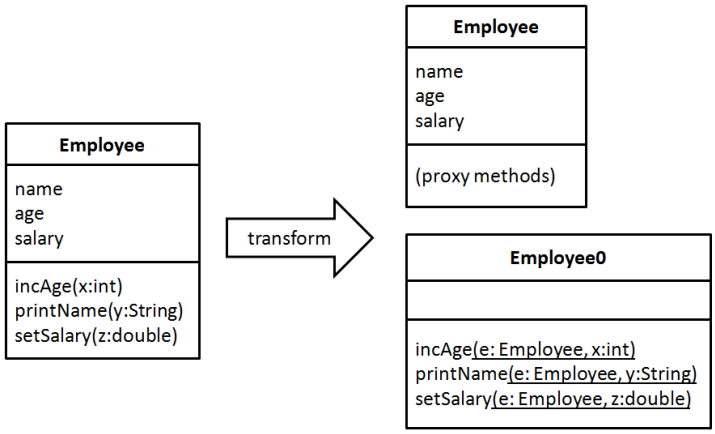

Figure 2.1: *JRebel class transformation.*

In the following, we will illustrate how different DSU systems modify the original code of a program to make it dynamically updatable. Figure 2.1 shows

a sample of JRebel class transformation. As it is shown in the sample, JRebel builds another class called `Employee0` and moves all of the `Employee` methods to it. JRebel makes methods static and adds a parameter of type `Employee` as the first parameter to them. The receiver object is passed to each method as a first argument. Each method on `Employee` uses JRebel APIs and lookups the current implementation of the corresponding method and calls it. For the fields, *ZeroTurnaround*[1] does not disclose the details of JRebel internals beyond the filed patents [71].

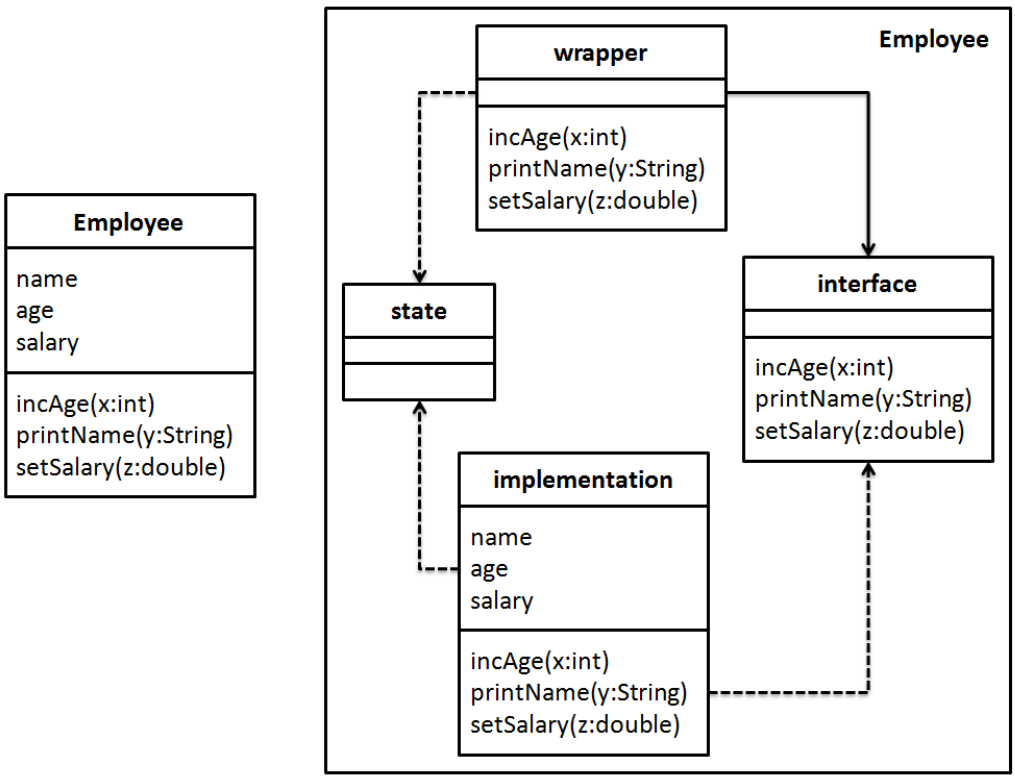

Figure 2.2: *DUSC class transformation.*

Figure 2.2 presents DUSC class transformation for `Employee`. DUSC converts each class C into the four individual classes:

1. **Implementation class**. C_i contains the implementation of version i of class C. It includes both fields and methods of original C with slight modification. Furthermore, it holds a reference to the corresponding wrapper class and a method to return status in the state class.

[1] `https://zeroturnaround.com/`

2. **Interface class**. C_a is an abstract class that all the different versions of class C should implement it. This class is exploited as a static field inside the wrapper class to call methods on the implementation class. When the implementation class is changed, the new implementation class is still a subclass of C_a and the wrapper class can refer to it through the same code.
3. **Wrapper class**. C_w provides the same interface that class C provides to any client class of C. For each method m of C, there is a corresponding method m_w in C_w with the same signature. m_w can invoke the current version on m in the implementation class.
4. **State class**. C_s contains the same fields of C_i and its objects are employed to migrate state of the old instances of implementation class to the new versions.

After introducing Java version 1.4, most of the JVMs are equipped with HotSwap which allows running applications to change method's body dynamically. However, some DSU systems benefit this poor capability. By taking this ability, JavAdaptor provides dynamic future changes for programs through containers and proxies. Figure 2.3 demonstrates an example of JavAdaptor class transformation. As it is shown, a `container` field is added to the initial code 2.3a at line 6. The `averageTemp` method at line 2 is replaced with `currentTemp`. JavAdaptor changes class `TempSensor` name into `TempSensor_2` to avoid name clashes. Moreover, it generates two classes as a `container` and `proxy` classes. HotSwap replaces the method's body with the code which contains proxy and container codes.

2.4 Challenges

Regardless of the mechanism used in the DSU system, some issues should be considered. Some of them are essential parameters, while some others are a relative subject that is comparable in different systems. Obviously, some systems pay more attention to some parameters and ignore others. Moreover, there are some contradictory issues that fulfilling one of them will lead to another violation. Nevertheless, the systems try to strike a balance between them. In the following, some of the key issues in these systems are briefly explained.

```
class TempSensor {
  float averageTemp() { ... }
}
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  void displayTemp() {
    ts.averageTemp();
    ...
  }
  TempSensor getSensor() {
    return ts;
  }
 }
```

(a) *initial code.*

```
class TempSensor_v2; {
  float currentTemp() { ... }
 }
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  void displayTemp() {
    ((Container) cont).ts.currentTemp();
    ...
  }
  TempSensor getSensor() {
    return new Proxy((Container) cont).ts ;
  }
 }
 // Generated for this update
 class Container implements IContainer {
  TempSensor_v2 ts;
 }
 class Proxy extends TempSensor {
  TempSensor_v2 update;
 }
```

(b) *transformed code.*

Figure 2.3: *JavAdaptor class transformation.*

2.4.1 Flexibility

Java programs consist of classes. Each class internally includes a set of fields to hold its state and a set of methods to present its behavior. Externally, each class may relate to other class by inheritance and implement relationships. Each class has a single parent and may implement several interfaces. Flexibility in supporting various kind of changes is a major benefit to DSU systems. However, all DSU systems do not support every internal and external change.

For the internal changes, HotSwap only permits modifications on the body of methods. This is the least amount of flexibility among the DSU systems. However, most of the JVMs support this capability. DUSC is more flexible than HotSwap and allows the new version to add fields and methods as well as modify the existing fields and methods. Other DSU system such as JDrums, DVM, JVolve, the DCE-VM, JRebel, and JavAdaptor is quite flexible and support all of the internal changes. Nevertheless, some systems rewrite the original code of both old and new versions to eliminate the change. For example, JavAdaptor removes all final modifier of class members to make this change ineffective.

For the external changes, each class only inherits from one class. This makes the inheritance relationship as a tree. Adding a new class to the program is like adding a leaf to this tree. This change is supported by all of the DSUs. But a change in the inheritance reshapes the inheritance tree. JDrums, DUSC, and JVolve do not allow this change while JRebel, DVM, DCE-VM, and JavAdaptor support it. Although, this change may lead a type-safety violation. JavAdaptor prevents this violation by using proxy mechanism and modifies the method's body to refer the updated classes and applies this by exploiting the HotSwap.

2.4.2 Type Safety

In the statically typed language such as Java, every item's type should be determined at compile time and it should keep the same type during its lifetime. A program is type-safe if for each item in the program, all of its clients see it having the same type [114]. For example, an Integer item should not be interpreted as a Boolean and should not be used by Boolean operators. Type safety violation may occur in a DSU process due to changing an item type without updating the remaining part of the program using the item. This may lead to a runtime crash.

In general, DSU systems can preserve a program type-safe in the dynamic update process by employing four different techniques: ostrich, stub modules, update reordering, and type-checked updates. For example in stub modules

technique, the programmer provides a stub function for each modified module. This stub function adds an indirection level for the requests and redirects them to the appropriate version. All the DSU systems in Java try to propose a type-safe solution for dynamic update. However, in most of the DSU system, the code on the stack is not updated immediately and it may potentially lead a type-safety violation due to heterogeneous access to the updated item from the outdated items on the call stack.

2.4.3 Update Point

When the new version of the program is ready, the decision to apply the update to the running application is one of the issues in dynamic update systems. In general, three different times can be distinguished: i) The time to start update process; It usually includes loading the new classes and modifying the already loaded classes by using facilities such as HotSwap. Starting this process does not mean that the running application will immediately be affected by the new code. ii) The start time of the new code effect; It happens when the new code is used by the running application for the first time. Usually, the DSU systems do not update the call stack and the execution of the program continues with the old code on the stack. When a new invocation occurs, it is picked up from the new code. It may cause a temporary inconsistency due to running old and new code consequently. iii) Time to finish the update process. When all the old code is out of reach, the program completely switches to the new version and the update process is over. Normally, the first time is determined by the DSU system or the user, and the second and third times occur in sequence automatically. This time is known as an *update point* or *update moment*.

Update point can be determined in three different manners: i) DSU system asks the developer to specify the update point by inserting a method call inside the code. Each time that this method is invoked, the running application checks for the existence of the new patch. If a new patch is ready, the update process starts automatically. JVolve and DCE VM use this approach. ii) The update process immediately starts when a new update is ready. DSU system usually receives a command to start the update process. JRebel and JavAdaptor apply the new code instantly. iii) DSU system waits to satisfy certain constraints. It usually helps DSU system to avoid some inconsistencies which may occur during the update process. Some business considerations may also be involved.

2.4.4 State Transformation

State transformation is one of the main concerns in DSU systems. They need to transform outdated objects to their newly instances. An object state includes the values which are assigned to its fields. DSU should provide a mechanism to transform these values from old instances to the new ones. It is clear that the DSU system must be somewhat flexible to support changes in the fields, in order to make the state transition meaningful. For instance, the HotSwap only allows method's body modification. Therefore, it does not provide any state transformation mechanism.

There are two types of state transformation: automatic and assisted. In the automatic approach, the tool matches unmodified fields according to their names and types; and copies the values from old fields to the new fields automatically. New and modified fields can be initialized by the default values according to the default field initialization rules in Java programming languages [50]. Some of the DSU systems such as JRebel, DVM, and DCE VM only support automatic state transformation.

Automatic state transformation usually is enough for simply copying values of same fields between two versions. But initializing the added and changed fields with the default values is unrealistic and may lead to loss of program state and unusual behaviors. The developer can write own proprietary state transformation code to ensure correctness of transformation. Some systems such as JDrums, DUSC, JVolve, and JavAdaptor support assisted state transformation alongside automatic one. DSU system usually generates state transformation code automatically. However, the developer can alter this code and add specific semantic.

Regardless of whether the automatic or assisted approach is exploited to transfer the program state, each DSU system may follow a different policy for choosing the time to execute the state transformer. There are two different policies: immediate update and lazy update. Immediate approach transforms the state of the program together. JVM level DSU systems such as DCE VM and JVolve can take advantage of garbage collection algorithm to find outdated instances and transform their state while GC algorithm execution. Code level DSU system like DUSC and JavAdaptor detect outdated instances by different techniques. DUSC rewrites contractor and forces every instance to register itself in a vector object. In this way, state transform function can be run on all live instance when the program is paused.

Immediate approach increases the pause time during the dynamic update because all of the instances should be transformed together. The lazy approach provides a gradual state transformation and thus decreases pause time. Each

instance can be updated when it is accessed. For instance, JRebel adds a redirection call to each method's body. After performing an update, JRebel employs this redirection to detect the outdated instances and transform their state before using them. DVM and JDrums follow this policy to state transformation.

2.5 Validation

Aside from the challenges posed by DSU systems, one of the key challenges is how to make sure that the dynamic evolution process is done without introducing an error. Errors may stop the update process and produce a runtime crash. This is not acceptable for long-running programs which high availability is a critical property. Stopping the running program normally, replacing the code and re-execute program with the new code (cold update) is better than facing an error in the dynamic update process. In a volatile mode, the running program may confront with a temporary inconsistency and expose some wrong behaviors. Even this mode might not be acceptable on the most systems and causes semantic errors. Ensuring that the update process is performed without any runtime or semantic error is called *validation* or *correctness*.

Gupta et al. introduce the notion of update validity [59]. They prove that generally, it is undecidable to determine if a given arbitrary update is valid. First, they define reachability concept: a state s is said to be a reachable state of a program Π if and only if a process executing Π from its initial state $s_{\Pi 0}$ can reach s at some time for some inputs. Second, they define validity property: consider a program P is running code Π at the state s. A dynamic update process that applies a new code Π' to the running program P and transforms state s to s' using a state mapping function is valid if and only if after the update, P is guaranteed to reach a reachable state of P' in a finite amount of time. Practically the dynamic updated program should behave like a program which is running from the initial state using the new code. However, before the reaching to this point, the program may go to unexpected states.

The validity property is shown in Figure 2.4 informally. On top, two successive updates are shown from version n-1 to n and from n to n+1. On the bottom, the version n is never executed. Consider the validation of update from n to n+1 in the top part, it is valid if and only if there is a point in the program that two part of the Figure 2.4 cannot be discerned. It means a valid dynamic update from version n to n+1 should guarantee that program reaches a state that is reachable when version n+1 runs from the beginning.

Therefore, there is no general solution for validation problem in DSU systems. However, it is still possible to find some points in the program execution that

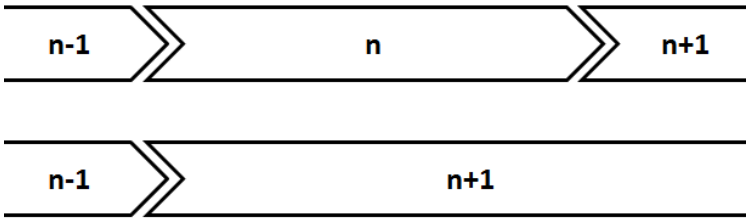

Figure 2.4: *Unofficial demonstration of DSU validation.*

```
File file;
void process() {
   ...
   read();
   ...
}
void read() {
   file = new File("sample.txt");
   FileInputStream stream =
       new FileInputStream(file);
           ...
}
```

(a) *before the update.*

```
File file;
void process() {
   ...
   file = new File("sample.txt");
   read();
   ...
}
void read() {
   FileInputStream stream =
       new FileInputStream(file);
           ...
}
```

(b) *after the update.*

Figure 2.5: *Example for update failure due to the choosing an unsafe update point.*

applying the update in those points are safe and the program continues its execution without any unexpected behavior or crashing. Let us explain the validation problem by an example. We will show how choosing an inappropriate update point can cause an error in the dynamic update process.

Figure 2.5 shows a Java piece of code for processing a file. The `process` method calls the `readFile` method. The `file` object is created and used by the `readFile` method in the old version. Instead in the new version, Figure 2.5b,

the `file` object creation is moved to the `process` method. Let us consider that the old version of the application before calling the `readFile` method (row 3 in Figure 2.5a) when the new code is replaced. In this case, the call to the `readFile` (row 4 in Figure 2.5a) 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 `process` method is completed. This example shows that even though both versions of a program run without error, choosing a wrong point to start the update process may cause the program to crash. In this thesis, we will show how we can predict the occurrence of these situations and dodge them at update time. In chapter 5 we will illustrate different approaches to the validation of DSU systems.

3
Validation Framework

The dynamic updating of running software systems cannot be realized without a specific support by the operating system, the programming language, or by an external middleware overlaying the language RTE/VM. These systems are described in chapter 2. The most critical property that any DSU system should respect is that the updating process must be performed without faults or errors and the software system must remain in a stable and sound state after the update. This issue, known as *Validation*, has not received the appropriate consideration by the DSU systems. We have a great deal to reach for a reliable and practical system. The validation issue is described in Section 2.5 and the various policies of different DSU systems in dealing with this issue will be illustrated in chapter 5.

Gupta et al. [10] formally proved that to determine if the deploying of an update on the running system will respect such a property is 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. In this chapter, we introduce a novel approach to provide a valid dynamic update for running applications.

First, we study how the dynamic update process can affect the execution path of a running program and how a fatal error may occur due to the choice of a wrong update point choice. Then we introduce a set of meta-data that can be exploited by the developer to express update constraints. In addition, we propose a validation framework that includes two parts: i) a static analyzer to determine unsafe update point. It processes old and new codes of a program, extracts unsafe parts of the code, and annotates them automatically ii) a runtime validator to find a safe update point by considering the defined constraints, include unsafe points which are produced by static analyzer automatically. Finally, to demonstrate the feasibility of the proposed approach we used it during the update of various distinct versions of three long-running system. The results of the experiments are reported and discussed.

3.1 Swinging Execution

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 is instantaneous, even if more and more often the state migration is unnecessary, such as in JavAdaptor [106]. 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 for 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.

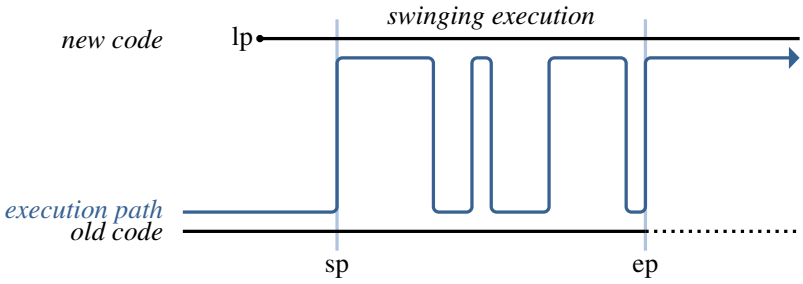

Figure 3.1: *Swinging execution.*

As shown in Figure 3.1, 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 is potentially inconsistent. This situation does not occur when either the old code or the new code are run separately.

This *transient inconsistency* [54] has been disregarded in some DSU systems [106, 51] 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 [117], 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 from that postponing the update is not always feasible or desirable, it may lead to endless waiting.

Another solution to this problem is to manually pop all active methods from the stack and push the updated version of them. This would prevent the execution of the old code after performing the update. Unfortunately, the JVM only support manually popping the methods and not pushing their counterparts updated version back to the stack. JVM does not support this feature because it can disturb the program's normal control flow and the running application is highly susceptible to producing unexpected behavior.

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 Figure 3.1). Starting the update at a safe point, the execution swinging cannot move the system in a *transient inconsistency* state. A *safe update point* [117] 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.

3.2 Annotation Driven Validation Process

The example of the Section 2.5 demonstrates that updating 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 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. To automatically determine a point in the application execution that drives to a valid update for any computation

without external hints is basically impossible [60]. As discussed in [114, 94], 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.

Each 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.

3.2.1 Proposed Annotations

The basic idea is to provide some facilities for developers to express these constraints. The easiest and most convenient way is to introduce these meta-data within the application code since their evaluation is a part of the application execution and of its updating. In Java, this means to use Java annotation facility. 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 facility of standard Java as well as developers are familiar with it. Limits and advantages of the Java annotation facility are analyzed in [29].

Annotations can be easily used to express both static or dynamic constraints. Although static constraints include some conditions that are specified before starting the deployment process, dynamic constraints depend on the status of the running application. For example, let us consider the scenario where the code in Listing 2.5a 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 be 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 of 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 reached. The developer is responsible for the correct selection of this point. This annotation can be exploited when the user wants to perform the update at a predefined point. However, the proposed static analyzer in Sec.3.4 may detect this point as an unsafe update point.

`@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 the developer can change this behavior and specify which threads should be affected.

3.3 Validator Component

According to the semantic relations between two versions of a program, the developer can decorate the program code with the proposed annotation. These constraints should be respected in the dynamic update time. We design and implement a runtime validator to verify these constraints before starting the update process. With respect to all of the specified restrictions, we can reach a safe update point to start deployment.

Existing DSU systems have a direct approach to software updating. The DSU

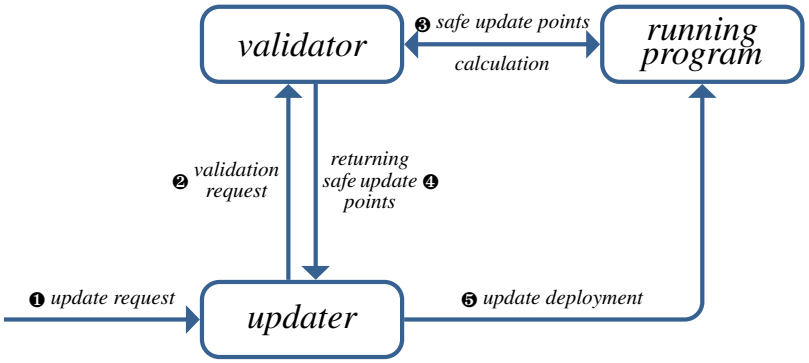

Figure 3.2: *Dynamic updating with validation.*

system, identified by the term *updater* in Figure 3.2, gets the requested updates (step 1 in Figure 3.2) and directly deploys them on the running software (step 5 in Figure 3.2) 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 Figure 3.2), calculates the safe update point thanks to the annotations in the code (step 3 in Figure 3.2), 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 Figure 3.2). The validator is an external and independent component and can be employed by any DSU.

Algorithm 1 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 for 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 [33]. 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

Algorithm 1: 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 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
```

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.

3.4 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 part of the 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 at a certain moment. Relevant to each simulation scenario are i) the point in the 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 reference 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 [68], PMD[1] and JLint [14][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.

3.4.1 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, the 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 points. To cope with this problem we 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

$$new = old + \Delta new - \Delta old + \Delta new' - \Delta old'. \tag{3.1}$$

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

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

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:

$$new = old + \Delta new - \Delta old + \Delta new' - \Delta old' + \boldsymbol{\Delta old^{+}} + \boldsymbol{\Delta old'^{+}}. \tag{3.2}$$

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:

$$\Delta old^{+} \vee \Delta old'^{+} \, refers \, \Delta old \vee \Delta old'. \tag{3.3}$$

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 access easily to the code still active and the calculation of the unsafe points is done statically. To respect this, what we know is the source code of the application before and after the change and consequently the extent of the change itself.Therefore, the described *refers* relationship must be relaxed to:

$$\Delta old \vee \Delta old' \, refers \, \Delta old \vee \Delta old'. \tag{3.4}$$

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

[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 mean 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.

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:

$$new_e = old + \Delta new - \Delta old + \Delta new' - \Delta old' + e. \tag{3.5}$$

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 Section 3.4.2) and when a problem is found the execution of the corresponding e is marked as an unsafe update point.

3.4.2 Technical 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 bodies 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` [44],

`Gum Tree` [41] and `Dependency Finder` [119]. 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 *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` [78] and `Spoon` [98]. 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 [98]. 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 interesting elements and the generation of the new annotated source code. The pseudo-code in Algorithm 2 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

Algorithm 2: Determining and annotating the unsafe update points.

$\{\Delta old \cup \Delta old'\}$ = `Dep. Finder` extracts changed/removed elements
foreach I_i *in* $\{\Delta old \cup \Delta old'\}$ **do**
  $\{e_i\} = \{e_i\}$ + *refers* I_i
foreach e_i *in* $\{e_i\}$ **do**
  $new_i = old + \Delta new - \Delta old + \Delta new' - \Delta old' + e_i$
  ΔPS_i = analyze new_i with `SPOON`
  if ΔPS_i *contains a problem* **then**
    mark e_i as unsafe

the application. Consider, for example, the Listing 2.5, 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 Section 3.3).

3.5 Evaluation

3.5.1 Considered Programs

In order to demonstrate the presented approach, we consider various versions of three different long-running programs and their dynamic evolution to the next version. 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 17 versions of three different applications to extend the variations in the update situations and to analyze how our approach behaves when the application size grows.

The first selected program is Hyper SQL Database (`HSQLDB`)[5] that its 1.* versions (the version number starts with 1) include about 370 classes. Six versions of this program have been selected that the change between two successive versions is not too much (1.8.0.9 to 1.8.1.3). In addition, we have selected two successive versions of this program from 2.* versions (2.3.2,2.3.3). These programs include almost 660 classes. As it is clear, the selected 2.* versions contain a large number of classes, as well as the difference between

[5] `http://hsqldb.org/`

two versions, is high enough. Other state-of-the-art DSUs exploit this program on own experiments[105, 100].

The second selected program is `CrossFTP` [6], a FTP server. We consider three versions of this program and their evolutions (1.07 to 1.11). These programs have about 230 classes. This program is also used in various DSUs to examine own systems [101, 117, 85].

The third program is Java Email Server (`JES`)[7], a simple SMTP and POP e-mail server, that we get it from the JES open-source repository. Versions 1.* of this program just have 20 classes. We consider 3 versions of this program (1.3,1.4,1.5). However, 2.* versions include almost 390 classes. We select 9 program from 2.* versions (2.5 to 2.9.0). Some DSUs have used this program in their experiments [128, 101].

We chose these applications for the following reasons. First, these programs need to service continuously and the dynamic update can be an ideal solution for upgrading these programs without interrupting their functionality. Second, all selected programs have been exploited in at least two previous DSU systems to complete their evaluations. Third, we easily obtained different successive versions of these programs on their repositories. Finally, these programs are compatible with Java 7 that most of the DSUs support their dynamic evolution. We tried to consider small and big changes in the versions in the selection.

3.5.2 Experiment Results

`Dependency Finder` has been used to extract the changes from one of the considered versions to the next version. Table 3.1 reports the found changes for the shift from the old version to the new version within various columns respectively. Programs are listed in the first column of the table. As you can see, the version numbers are written in the front of the name of each program and separated by ->. The rest of the columns demonstrate the statistics of different changes on the various items. 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 included in our work. As can be seen, the magnitude of the numbers in each row is usually proportional to the magnitude of the difference in the version numbers of the program in that row. For this reason, the numbers in rows 6 (HSQLDB 2.3.2 ->2.3.3) and 15 (JES 2.7.1 ->2.8.0)

[6] `http://www.crossftp.com/crossftpserver.htm`
[7] `http://javaemailserver.sourceforge.net/`

are larger than the numbers in the rows for the versions of the same programs. Obviously, the numbers for different programs are not comparable. We have included these two cases in our study to examine the impact of large changes on the update.

	Removed class/interface	*Added classes/interface*	*Modified classes/interface*	*Removed fields*	*Added fields*	*Modified fields*	*Removed methods*	*Added methods*	*Modified method signatures*	*Modified method bodies*	*Removed constructors*	*Added constructors*	*Modified constructor signatures*	*Modified constructor bodies*
Program old version ->new version	Changes													
HSQLDB 1.8.0.9 ->1.8.0.10	1	11	1	6	20	10	9	17	23	53	0	0	2	4
HSQLDB 1.8.0.10 ->1.8.0.11	4	0	10	0	0	2	0	181	0	26	0	0	0	2
HSQLDB 1.8.0.11 ->1.8.1.1	0	6	11	2	14	42	185	27	15	149	2	2	0	13
HSQLDB 1.8.1.1 ->1.8.1.2	0	0	0	0	0	2	0	0	3	5	0	0	0	0
HSQLDB 1.8.1.2 ->1.8.1.3	0	2	2	0	2	1	0	14	0	34	0	0	0	5
HSQLDB 2.3.2 ->2.3.3	0	5	5	44	80	203	118	225	181	779	18	15	3	47
CrossFTP server 1.07 ->1.08	0	0	0	1	1	28	1	2	0	25	0	0	0	29
CrossFTP server 1.08 ->1.09	6	3	0	1	5	28	2	15	0	53	0	1	1	31
CrossFTP server 1.09 ->1.11	0	5	0	6	11	1	1	7	2	22	0	0	0	3
JES 1.3 ->1.4	0	0	0	0	10	1	2	11	1	12	0	0	0	1
JES 1.4 ->1.5	0	1	0	0	3	2	1	1	1	64	0	0	0	3
JES 2.5 ->2.6	2	9	8	23	13	17	15	34	5	120	7	7	2	30
JES 2.6 ->2.7.0	9	0	0	1	0	0	0	1	0	6	0	0	0	0
JES 2.7.0 ->2.7.1	2	5	4	0	4	2	2	4	4	43	0	0	0	2
JES 2.7.1 ->2.8.0	49	102	18	42	65	157	110	152	55	313	64	66	7	51
JES 2.8.0 ->2.8.1	0	0	1	7	2	2	11	11	0	13	2	2	0	7
JES 2.8.1 ->2.8.2	0	0	0	11	1	0	1	0	0	47	0	0	0	9
JES 2.8.2 ->2.9.0	0	8	1	5	6	1	2	13	4	44	7	7	0	5

Table 3.1: *Considered programs changes.*

Then for every removed/changed element, the elements (mainly methods and constructors) that refer to it are extracted as well. The numbers of these executables (methods or constructors) are listed in the second column of Table 3.2. The numbers in this column for each program represent the number of executables that have been modified or referenced to a changed/removed item.

It is important to note that the DSUs that follow a conservative policy in the dynamic update process consider the same number of executables as active methods/constructors.

Program	Considered executables	Unsafe executables	Improvement compared to the conservative approach (%)
HSQLDB 1.8.0.9 ->1.8.0.10	119	33	72.3
HSQLDB 1.8.0.10 ->1.8.0.11	52	0	100.0
HSQLDB 1.8.0.11 ->1.8.1.1	401	45	88.8
HSQLDB 1.8.1.1 ->1.8.1.2	11	0	100.0
HSQLDB 1.8.1.2 ->1.8.1.3	43	0	100.0
HSQLDB 2.3.2 ->2.3.3	1340	256	80.9
CrossFTP server 1.07 ->1.08	55	1	98.2
CrossFTP server 1.08 ->1.09	94	3	96.8
CrossFTP server 1.09 ->1.11	30	9	70.0
JES 1.3 ->1.4	17	5	70.6
JES 1.4 ->1.5	70	4	94.3
JES 2.5 ->2.6	204	50	75.5
JES 2.6 ->2.7.0	15	0	100.0
JES 2.7.0 ->2.7.1	65	24	63.1
JES 2.7.1 ->2.8.0	747	307	58.9
JES 2.8.0 ->2.8.1	35	20	42.9
JES 2.8.1 ->2.8.2	57	39	31.6
JES 2.8.2 ->2.9.0	68	33	51.5
Avrage			82.1

Table 3.2: *Results of experiments.*

Then all the elements (methods and constructors) that refer to a changed element are checked. According to the approach presented in Section 3.2 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 call

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` employs Eclipse compiler that can detect 626 different kinds of potential problems divided into 16 categories. The names and explanations of these categories are listed in Table 3.3 [2].

As shown in the description of the problem categories in Table 3.3, 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 classified based on problem type and reported in Table 3.4. For better representation and avoidance of a dense table, categories that have no problems are removed. The numbers in the table represent the number of distinct executables that cause the problem, not the total number of problems. This means that the examination of some cases may lead to the production of more than one problem from a specific category. Also, in some case studies, several problems may arise from different categories. Therefore, the numbers are overlapping in a row, and their sum does not present the number of distinct problematic items. The number of items that have fatal problems and are annotated as an unsafe code by the static analyzer is shown in column 3 of Table 3.2.

3.5.3 Discussion

The analysis in our demonstration study is limited to the fatal errors. Optional problems cannot create a serious runtime error in the program. As an instance, column 5 of Table 3.4 shows the category of problems that related to the coding style. This kind of problems is well-known as a *code smell or bad smell*

Category Name	Description
BUILD PATH	Problems related to buildpath
CODE STYLE	Optional problems related to coding style practices
DEPRECATION	Optional problems related to deprecation
IMPORT	Fatal problems in import statements
INTERNAL	Fatal problems which could not be addressed by external changes, but require an edit to be addressed
JAVADOC	Optional problems in Javadoc
MEMBER	Fatal problems related to type members, could be addressed by some field or method change
NAME SHADOWING CONFLICT	Optional problems related to naming conflicts
NLS	Optional problems related to internationalization of String literals
POTENTIAL PROGRAMMING PROBLEM	Optional problems related to potential programming flaws
RESTRICTION	Optional problems related to access restrictions
SYNTAX	Fatal problems related to syntax
TYPE	Fatal problems related to types, could be addressed by some type change
UNCHECKED RAW	Optional problems related to type safety in generics
UNNECESSARY CODE	Optional problems related to unnecessary code
UNSPECIFIED	List of standard categories used by Java problems, more categories will be added in the future.

Table 3.3: *Eclipse compiler problems categories.*

[121]. Code smells usually are not a bug and they do not interfere the normal execution of a program. Thus, the optional errors are ineffective and ignored in this study.

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. As already mentioned, these two categories are not listed in Table 3.4 because of their zero numbers for all rows.

The `TYPE` kind of errors 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 on HSQLDB 2.3.2 ->2.3.3 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 3.3a with the new in Listing 3.3b.

Another example of this kind of problems occurs in JES 2.7.1 ->2.8.0. The constructor of class `com.ericdaugherty.mail.server.configuration.TreeAttribute` is declared with a `throw Exception`. The callers of this constructor in the class `com.ericdaugherty.mail.server.configuration.cbc.ApplyConfiguration` deal with the exception using try/catch block or add an identical throw (for the same exception or a supertype) statement to the caller declaration. In the new version `throw Exception` is eliminated from the constructor declaration. Obviously, the callers are modified to adapt to this change and do not deal with the exception. Thus, the temporary code that is composed of the new code and the old constructor has a compiler error: «Unhandled exception type Exception».

As shown in Table 3.4, the `MEMBER` column has the largest number of problems than the other columns. The `MEMBER` kind of errors usually occurs when an element tries to access to a removed or modified element. For example, in CrossFTP server 1.08 ->1.09, it occurs when in the new version of the `org.apache.ftpserver.gui.ServerFrame` class, the `FEEDBACK_PAGE` field has been removed and the method `doFeedbackCommand()` tried to access it and the error «`FEEDBACK_PAGE` cannot be resolved or is not a field» is occurred.

This kind of problems also occurs when the the visibility of an element is reduced in the new version. For instance, the visibility of method `createDomainDirectory(java.lang.String)` in class `com.ericdaugherty.mail.-`

```
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.*

Figure 3.3: *Example for the* `TYPE` *problem.*

`server.configuration.ConfigurationManagerDirectories` in project jes-2.8.2 ->2.9.0 is changed from public to private. Therefore, it is not accessible from method `createDomainDirectory` in class `com.ericdaugherty.mail.server.configuration.ConfigurationManager` and find-bug tool reports this problem: «The method `createDomainDirectory(String)` from the type `ConfigurationManagerDirectories` is not visible». Similar thing may occur in reducing the visibility of fields. For instance, in HSQLDB 2.3.2 ->2.3.3, when the `initParams` method in the `org.hsqldb.persist.Log` class tries to access a field which is not visible in the new code, this error is created: «The field `Logger.propLogSize` is not visible». Because the visibility of `propLogSize` is changed from public to package.

When a final modifier is added to a field declaration, two `MEMBER` problems may be reported: First, the final field is not initialized in the temporary code. Second, the final field is illegally assigned in the code. In the HSQLDB 2.3.2 ->2.3.3, when `rowId` field from `org.hsqldb.RowAction` gets a final modifier in the new version, find-bug tool reports two errors: «The blank final field `rowId` may not have been initialized» and «The final field `RowAction.rowId`

cannot be assigned». However, adding a final field may not make a problem in the temporary code. For instance, `mainBlockSize` field from `org.hsqldb.persist.TableSpaceManagerBlocks` gets a final modifier, but it does not make a problem. Because it is initialized one time in the constructor and never assigned in the other parts of the code.

Removing the static modifier from an element may also report `MEMBER` problem. In JES 2.5 ->2.6, when field `watcher` of class `com.ericdaugherty.mail.server.configuration.ConfigurationManager` changed from static to a non-static field in the new version, the bug finder reveals this error: «Cannot make a static reference to the non-static field `com.ericdaugherty.mail.server.configuration.ConfigurationManager.watcher` ». Because static method `shutdown()` tries to access to this non-static field from the old code. The similar thing may happen for the static methods. In JES 2.7.1 ->2.8.0, static method `getPassReceivedLocalMessage()` in the class `com.ericdaugherty.mail.server.configuration.ModuleControl` is changed to be a non-static method in the new version. The bug finder detects a problem in the temporary code because method `deliverLocalMessage` in the class `com.ericdaugherty.mail.server.services.smtp.client.SMTPSenderStandard` in the old version may invoke this method through the class, not from instances. This problem is reported by this message: «Cannot make a static reference to the non-static method `getPassReceivedLocalMessage()` from the type `ModuleControl`». However, if the changed method is never invoked statically by the old code, removing static modifier in the new version cannot make a problem in the temporary code. For instance, in JES 2.5 ->2.6, static modifier is removed from method `setPassword` in class `com.ericdaugherty.mail.server.configuration.LoginCallbackHandler`. This method is invoked through the instances in the old code. Thus, removing the static modifier cannot produce a compile problem.

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 experiments. An instance of this kind problems is occurred in the method `getTableSpace` in the `org.hsqldb.persist.DataSpaceManagerBlocks` class in HSQLDB 2.3.2 ->2.3.3. This error message is generated: «The operator >= is undefined for the argument type(s) `int`, `AtomicInteger`». Because 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 3.4a with the new in Listing 3.4b.

```
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.*

Figure 3.4: *Example for the* `INTERNAL` *problem.*

Along with these problems, there are some obvious rules that we expect to be met. The first fact is about deleted items. We expected that all the elements that refer to removed methods and constructors 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 tries to access these items it will provoke a compilation error. However, we found some cases that violate this rule. 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 these 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, in HSQLDB 2.3.2 ->2.3.3, 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.

Another fact is about the changed methods and constructors. 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.PersistentStoreCollectionDatabase` has a parameter of type `java.lang.Object` which has been changed to `org.hsqldb.Ta-`

`bleBase` 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. 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. However, it cannot be generalized since it depends on the type of changes.

The summary of the results is shown in Table 3.2. The first column lists the case studies. The second column shows the number of examined executables (methods and constructors). These numbers show the number of executables may be active at the update time. Remember 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), it should wait until none of these active methods is on the stack to start the dynamic update process. The third column presents the number of executables that are detected as a problematic item in the static analysis process and are marked as unsafe. The difference between the numbers in the second and third columns indicates the number of active executables that are safe to start the update. This means that if these items are in the call stack, the update process can be performed. Thus, the dynamic deployment can be free of unnecessary delay. A lower number of unsafe update points implies also a more agile update process with less constraint to satisfy. The improvement percentage is shown in the fourth column. The improvement rate is from 31% to 100%. The average improvement rate for the 17 case studies is 82.1%. However, for three cases this number is 100. It means the dynamic update process can be started immediately.

3.5.4 Time Information

Although detection of unsafe update points and annotating them are performed statically before the start of the update process, it is useful to have an imagination of the time it takes to run this process.The summary of these measurements is presented in Table 3.5. Measurements were carried out on a machine with

Intel Core i3 CPU (M 330 @2.13GHz 2.13GHz) and 4GB of RAM running Windows 7 64-bit Operating System. We used the Oracle JVM version 1.7.0_55 with HotSpot 64-Bit Server VM (build 23.55-b03) to execute applications. The experiments are repeated 3 times and the average measured times are written in the table. Method `System.currentTimeMillis()` is used to get the times.

The first column of Table 3.5 lists the programs and their successive versions. To have information about the program size, the second column demonstrates the size of the programs in *Line Of Code (LOC)*. The sizes of both versions are calculated because both of them are included in the process. The sizes vary from 10k LOC to 598k LOC.

The program analysis to extract unsafe points consists of two steps. The first step is to build the `Spoon` model of the old and new versions of the program. As mentioned earlier, we have used the `Spoon` library to process the source code. `Spoon` needs to create a meta-model of program source code in memory. This process is time-consuming and is done one time for both versions. The measured times for this step are in the third column. As you can see, this number is about 2 seconds for the small programs such as JES 1.4 ->1.5 and for the big programs such as HSQLDB 2.3.2 ->2.3.3 is about 241 seconds. If these numbers are compared to the program sizes, it can be concluded that there is a direct relation between the time of construction of the model and the size of the programs. The ratio of the time for building the model to the size of the programs is given in column 7. This ratio varies from 157 to 404 for different case studies.

The second step is to inject an executable into the new code, create a temporary code, examine this code, and annotate the executable if there is a problem on examination. The measurement time for this step is presented in column 4. This process is repeated for all of the extracted items. Again, by comparing these numbers with the numbers in column 2, there is a direct relation between the time needed for the temporary code processing and the program size. In fact, there is no significant difference between the sizes of the temporary code and the new code. Therefore, this proportion is logical. These ratios are shown in column 8 of the table. The range of numbers varies from 10 to 23.

The total time required for processing is obtained from the following formula:

Total process time = Model creation time +
*(Finding bugs time for each item * Number of items)*

The numbers for this calculation are shown in column 6. As predictable, the total time is more related to the second stage. Although the processing time of

each item is not so high, the repetition for each item increases the total time. Therefore, in the cases that the number of modified items and program size are large, the total time for processing would be high. Although this value is only 6 seconds for small programs with modest changes, such as JES 1.3 ->1.4, it takes almost 3 hours for a large program with many changes, such as HSQLDB 2.3.2 ->2.3.3. Please note that this static analysis can prevent the occurrence of fatal errors at the dynamic update time. Also, compared to DSUs that follow conservative policy, the wait time for stating dynamic update process is almost reduced up to 82%. Moreover, the experiments are executed on not so strong machine with a limited memory size that may cause virtual memory usage on loading and on processing large programs. It may increase process time.

3.6 Summary

Within this chapter, we presented our approach for validating a dynamic update process. We proposed a framework with two parts: i) A static analyzer to process two versions source code and determine unsafe parts of the code. If the dynamic update process starts when the program is executing an unsafe code, the program will be crashed and the update process will be failed. Dynamic updater should wait until all of the unsafe codes are removed from the call stack. ii) A run-time component is implemented to support this idea at the update time. This component dodges the unsafe update points by scrolling the call stack of running program and informing the dynamic updater after finding the call stack empty of unsafe points. We demonstrated the applicability of this method by applying this to various versions of three different applications.

Program	TYPE	MEMBER	INTERNAL	CODE STYLE	POTENTIAL PROGRAMMING PROBLEM	DEPRECATION	UNNECESSARY CODE	UNCHECKED RAW
	Problem categories							
HSQLDB 1.8.0.9 ->1.8.0.10	1	32	0	4	24	0	5	25
HSQLDB 1.8.0.10 ->1.8.0.11	0	0	0	0	0	0	0	0
HSQLDB 1.8.0.11 ->1.8.1.1	1	44	0	0	2	0	6	1
HSQLDB 1.8.1.1 ->1.8.1.2	0	0	0	0	0	0	0	0
HSQLDB 1.8.1.2 ->1.8.1.3	0	0	0	0	0	0	0	0
HSQLDB 2.3.2 ->2.3.3	5	252	1	8	19	0	118	36
CrossFTP server 1.07 ->1.08	1	0	0	1	1	0	2	1
CrossFTP server 1.08 ->1.09	1	2	0	2	2	0	3	2
CrossFTP server 1.09 ->1.11	0	9	0	0	0	0	1	0
JES 1.3 ->1.4	2	3	0	1	0	0	0	2
JES 1.4 ->1.5	4	0	0	0	0	0	1	4
JES 2.5 ->2.6	4	48	2	0	4	0	4	5
JES 2.6 ->2.7.0	0	0	0	0	0	0	0	0
JES 2.7.0 ->2.7.1	3	21	0	0	0	0	13	16
JES 2.7.1 ->2.8.0	117	231	2	0	3	0	61	98
JES 2.8.0 ->2.8.1	0	20	0	0	0	0	1	0
JES 2.8.1 ->2.8.2	0	39	0	15	0	0	10	16
JES 2.8.2 ->2.9.0	2	33	0	5	2	15	2	17

Table 3.4: *Classification of detected problems in inspected codes based on error type.*

Program	LOC(k)	Model creation time (ms)	Find bugs and annotating time for each item (ms)	Number of cases	Total process time (s)	Model creation time/LOC (ms/k)	Find bugs and annotating time for each item/LOC (ms/k)
HSQLDB 1.8.0.9 ->1.8.0.10	310	93365	3341	119	491	301.18	10.78
HSQLDB 1.8.0.10 ->1.8.0.11	314	89743	3429	52	268	285.81	10.92
HSQLDB 1.8.0.11 ->1.8.1.1	314	90105	3321	401	1422	286.96	10.58
HSQLDB 1.8.1.1 ->1.8.1.2	314	89107	3364	11	126	283.78	10.71
HSQLDB 1.8.1.2 ->1.8.1.3	316	92774	3166	43	229	293.59	10.02
HSQLDB 2.3.2 ->2.3.3	598	241751	8407	1340	11507	404.27	14.06
CrossFTP server 1.07 ->1.08	62	12177	1274	55	82	196.40	20.55
CrossFTP server 1.08 ->1.09	62	9783	1262	94	128	157.79	20.35
CrossFTP server 1.09 ->1.11	62	10079	1252	30	48	162.56	20.19
JES 1.3 ->1.4	10	2859	207	17	6	285.90	20.70
JES 1.4 ->1.5	10	1821	203	70	16	182.10	20.30
JES 2.5 ->2.6	109	21042	2392	204	509	193.05	21.94
JES 2.6 ->2.7.0	108	19105	1858	15	47	176.90	17.20
JES 2.7.0 ->2.7.1	107	19447	2242	65	165	181.75	20.95
JES 2.7.1 ->2.8.0	105	18562	2466	747	1861	176.78	23.49
JES 2.8.0 ->2.8.1	102	18320	2415	35	103	179.61	23.68
JES 2.8.1 ->2.8.2	102	19104	2230	57	146	187.29	21.86
JES 2.8.2 ->2.9.0	102	17568	2160	68	164	172.24	21.18

Table 3.5: *Time information about the automatic annotating process.*

4
Dynamic Updating and Code Smells

The term of *code smell or bad smell* usually refers to any symptom in the program source code that probably portends a deeper problem[121]. It also involves some certain structures that violate fundamental design principles and have negative impacts on the design quality of the program[118]. Code smells usually are not a bug and they do not interfere the normal execution of a program. However, they may cause other problems such as performance penalty or increase the risk of bugs or failure in the future. For instance, *Spaghetti Code* as a code smell does not affect the program execution but increases the maintenance cost and risk of failure in the future.

Several books have been written on smells. Flower[118] introduced 22 code smells such as long methods, duplicated code, large class, and long parameters. Code smells are described informally and also some refactoring techniques are proposed. Smells can be considered from different aspects. Webster [122] presents smells in the object-oriented programming, involving conceptual, political, coding, and quality assurance pitfalls. Therefore, the problems that may occur due to dynamically update of a program can be considered as a criterion for determining code smells.

In the previous chapter, we showed that swinging execution as an inevitable phenomenon in the dynamic updating process. It is common in the most of the DSU systems and may cause transient inconsistency during or after the update. Although the program may pass from this phase safely, in a pessimistic scenario, this transient inconsistency may lead to a fatal error and update failure. This is a disaster for long-running programs. As it is mentioned in Sec.3.4, The behavior of programs in dealing with this phenomenon is predictable. The unsafe points of the code can be annotated statically through an automatic process and can be dodged at the update time to choose a safe update point. However, the impact of various changes in this phenomenon is not entirely clear. Some types of changes on the program can not have a negative effect on its

dynamic updating. Studying the impact of each atomic change in the program dynamic update can extract the smells on the program code, regarding the dynamic update issue. These smells can be considered in the development process.

In this chapter, we will first develop a set of candidate error-prone patterns with respect to the language features and their possible atomic change. Patterns are associated with atomic changes that may occur in program evolution.These patterns are used to explore code smells on the dynamic software updating. In addition, this set can be exploited as a reference set by other DSU tools to measure flexibility. To explore smells, we examine the dynamic evolution of candidate error-prone patterns on at least three state-of-the-art DSUs and categorize the results. We consider fatal errors that may crash program as well as syntax errors that violate language rules. Moreover, the candidate patterns are examined by our proposed static analyzer to explore heterogeneities results. Finally, we trace these situations and enhance the static analyzer to cover this defect and disclose all of the code smells.

4.1 Building Error-prone Patterns

Obviously, the updating process mainly consists of changing the code of the application to update. Basically, in order to understand what could go wrong during the updating process, it is necessary to know how the code could change and when such a change would drive forth to a fatal situation. Such a level of awareness could be difficult to achieve especially if you are considering it at the level of the application code since the code could be really complicated and the provenance of the error could be tangled.

A more limited (but still meaningful) awareness can be achieved by considering the single language features independently of how these are used in the application code. Given a single language feature, it is possible to build a simple sample program, called *error-prone pattern*, that manifests a problem during the updating process when the update affects the considered language feature. Therefore an error-prone pattern is characterized by both the language feature and how such a feature is changed. Considered language features are the class, interface, enum and annotation as well as their possible members such as field, method and so on. Possible modifications include adding and removing language features, altering the language feature characteristics (e.g., visibility, type, ...) and how the language features interact (e.g., nesting, hierarchy, ...). The execution of these error-prone patterns on different DSU frameworks will provide us a deep and clear understanding of how these DSU frameworks

behave in the context described by the error-prone patterns.

	language feature	1 *add*	2 *remove*	3 *type change*	4 *value change*	5 *visibility change*	6 *add/remove static*	7 *add/remove final*	8 *add/remove abstract*	9 *add/remove synchronized*	10 *add/remove transient/volatile*	11 *change implemented interfaces*	12 *change inheritance*	13 *move to sub/superclass*	14 *change nesting*
		modifications													
1	class	⊘	●			●		●	●			●	●		
2	interface	⊘	●			●			⊘				●		
3	inner class	⊘	⊘			⊘	●	⊘	⊘			⊘	⊘	●	●
4	method in class	⊘	●	●		●	●	●	●	●				●	
5	class constructor	●	●	●		●									
6	method in interface	⊘	●	●					⊘					●	
7	field in class	⊘	●	●	●	●	●	●	●		●			●	
8	field in interface	⊘	●	●	●		●	●						●	
9	enum	⊘	●			●									
10	enum constant	●	●	●	●										
11	enum method	⊘	●	●		●	⊘	⊘	⊘	⊘					
12	annotation	⊘	●	●	●				●				●		

Table 4.1: *Possible changes in the program.*

Table 4.1 represents the full set of language feature/modification combinations for Java 7.[1] Each row reports a language feature and each column the possible changes. Of course, not all the combinations make sense. An empty cell indicates that the combination is inapplicable in the Java language, e.g., to change the value of a class. A ● symbol at a combination indicates that the changing in the language feature can occur whereas a ⊘ symbol still indicates that the combination can occur but this is not further considered either because they are covered by other combinations—e.g., removing an inner class is technically equivalent to remove a class—or, even if, this combination would occur the changed code could not be used from the old code—e.g., any newly introduced method cannot be invoked by the old unchanged code; note that a new constructor without parameters is implicitly called when the class is instantiated even if this is not changed yet. Of course, any code (includes

[1]We chose Java 7 because no DSU supports Java 8 at the time of writing.

the new code) can be used by old code through Java reflection. But it never happens practically in the real applications. Furthermore, our static analyzer detects unsafe part of the code by employing bug-finder tools which cannot find problems in the Java reflection codes. So, Java reflection considering is useless in our technique. However, our runtime validator can detect the unsafe executables that are invoked through Java reflection because it scrolls the call stack that involves all of the invoked executables even through reflection. As can be seen, for sake of readability, the columns with the same values have been merged in the table.

In defining the samples the following criteria were considered: i) The cases are developed as simple as possible to cover only one cell of the table ii) Two versions of samples can run individually without any error iii) Cases are independent of the DSU tools that perform the update on them iv) The update process is done out of the `main` method because most of the DSU tools have a problem with updating it v) The cases are developed in a way which is closer to make a potential fault. For instance, in the case of visibility change of an item, reducing the visibility is more prone to make an error. So, we have examined the impact of reducing visibility not increasing. However, in most cases, all possibilities have been examined.

The general structure of the samples is based on the concept of swinging execution. The old code should have an executable (method, constructor) which contains a changed item. Exactly before the reaching the changed item, the DSU tool performs the update and program switches to the new version. The current executable continues the execution of the old code but each access should be picked up from the new code. This situation can create a misbehavior or runtime error.

Let us look at a sample that a *static* modifier of a method is removed in the new version of the code. As you can see in List 4.1, static modifier is eliminated from method `foo()` signature. Method `foo()` is invoked statically inside method `bar()` in the old version, but it is called through an object in the new version. Exactly before the second call, the program is updated to the new version. From this moment, the statically invoking of method `foo()` is not valid anymore. However, the program continues its execution inside the old version of method `bar()` and it creates a runtime error at line 8. So, performing this sample on the DSUs creates a runtime error.

In most experiments, we have followed the above scenario. Nevertheless, that is not the only pattern which is followed. For instance, in adding/removing `synchronized` modifier to a method, by making particular changes in the new version and selecting a specific point for performing the update, the program

```
class Sample{
  static void foo(){
    ...
  }
  void bar(){
    Sample.foo();
    doUpdate();
    Sample.foo();
  }
  public static void main(String[] args) {
    new Sample().bar();
  }
}
```

(a) *old code.*

```
class Sample{
  void foo(){
    ...
  }
  void bar(){
    new Sample().foo();
    //doUpdate();
    new Sample().foo();
  }
  public static void main(String[] args) {
    new Sample().bar();
  }
}
```

(b) *new code.*

Figure 4.1: *General form of error-prone patterns in the dynamic update.*

is stuck in a deadlock situation at the dynamic update process. This pattern is shown in List 4.2. As it is shown, the old code 4.2a contains three classes. `ClassA` has two methods with a synchronized modifier. `ClassB` also has two methods that only one of them has synchronized modifier. The `methodA` and `methodB` have the parameter which obtains an object from `ClassB` and `ClassA`. If the `methodB` also has synchronized modifier, the program is going to be stuck in a deadlock situation. In the new code, synchronized modifier is removed from `methodA` and added to `methodB`. If the old or new version of the program runs individually, none of them is faced with a deadlock situation. Imagine the old program is running inside `methodA` and exactly before calling `methodB`, the program is updated and `methodB` is picked up from the new version which has synchronized modifier. In this case, two objects waiting for each other and they never go out from this situation.

The basic idea is trying to put program execution in a direction that never happens in the old neither in the new version but mixing two codes may raise a runtime error. There is no claim on completeness of candidate patterns

```
class  ClassA {
  synchronized void methodA( ClassB b)
  { b.last();}
  synchronized void last()
  {  ...  }
}
class  ClassB {
  void methodB( ClassA a)
  { a.last();}
  synchronized void last()
  {  ...  }
}
class Deadlock implements Runnable
{
   ClassA a = new  ClassA();
   ClassB b = new  ClassB();
  Deadlock()
  {Thread t = new Thread(this);
    t.start();
    a.methodA(b);}
  public void run()
  { doUpdate();
    b.methodB(a);}
  public static void main(String args[] )
  { new Deadlock();}
}
```

(a) *old code.*

```
class  ClassA {
  void methodA( ClassB b)
  { b.last();}
  synchronized void last()
  {  ...  }
}
class  ClassB {
  synchronized void methodB( ClassA a)
  { a.last();}
  synchronized void last()
  {  ...  }
}
class Deadlock implements Runnable
{
   ClassA a = new  ClassA();
   ClassB b = new  ClassB();
  Deadlock()
  {Thread t = new Thread(this);
    t.start();
    a.methodA(b);}
  public void run()
  {
  b.methodB(a);}
  public static void main(String args[] )
  {new Deadlock();}
}
```

(b) *new code.*

Figure 4.2: *Pattern of deadlock occurrence in the dynamic update.*

collection. However, various errors can be detected due to the performing an update in an unsafe point. The final point of this section is that the evaluation process is done manually rather than using automatic techniques like *Junit* test [45]. Because DSU tools usually do some modifications on original bytecode of the program to have dynamic update capability. These modifications may affect the automatic test part of the project and the results will be different. By considering above issues, we write 75 individual patterns.

4.2 Parameters Considered

After defining the candidate error patterns, all of them are applied to three different DSUs. The following parameters are considered in investigating the samples.

Support by DSU. The first thing that should be determined is whether a specific change is supported by the DSU or not? To answer this question, apart from what is mentioned in the literature, we follow a black box policy. In most cases, Java reflection helps us to make sure that the changes are applied correctly. For instance, when a field gets static modifier, these changes can be detected by the Java reflection. Other changes can be directly investigated by the user from the DSU log and program output. For instance, when a field's value is changed, it can be easily identified by printing the field value.

Runtime error in the dynamic update process. While a program is updating by a DSU tool, the worst scenario is that the running program is crashed by facing a runtime error. So, considering this, three modes may occur: i) Update is done correctly without any runtime error. ii) The program ends with throwing an exception and a runtime error occurs. iii) There is another mode that facing with a runtime error is uncertain and depends on other conditions. For instance, when a class is deleted from an application, even after the update, the old class is accessible from the old code. But if the class is not used before the update, it does not exist in the JVM, and any access to it from the old code will cause a runtime error. To eliminate these ambiguities, we separate different modes of these cases, study them one by one, and show them separately in the results. Finally, we consider the type of the exceptions in the case of runtime errors. All errors belong to the `java.lang` package. Thus, to avoid repeating, we delete the package name in the text and only write the class name.

Syntactic errors in the dynamic update process. Along with runtime errors, syntax errors may occur. These errors do not stop program execution but they are a violation of the rules of the programming language or a logical error. Three modes can be distinguished: i) when a program terminates with a runtime error, syntax error detection is meaningless. Therefore, in this case, we assume that no syntax error has occurred. ii) A syntax error happens. For instance, when visibility of a method is changed from public to private in the new version, obviously this method must not be accessible from the outside of the class. But after the update, it is observed that although the new code of the method is used, the program does not respect the visibility rules. This violates privacy in the programming language. These cases can be detected by investigating the program behaviors and outputs. iii) No unexpected behavior can be identified.

Errors detected by the framework. The last parameter considered in this study is which of the runtime and syntax errors found in the experiments can be detected by previously proposed validation framework. We also have five different modes here: i) A runtime or syntax error occurs and our approach can predict this error before starting the update process. ii) An error occurs during the update but the validator cannot determine this situation (false negative). iii) The validator identifies an unsafe part of the old code while the DSU can handle this situation properly (false positives). iv) An error does not occur and the static analyzer confirms that this case is safe. v) There is another mode that the static analyzer detects an error in a specific mode of the case study. For instance, in 5-3 when the signature of a class constructor which have parameters is changed, the validator marks this change as an unsafe if there is no implicit type conversion between the parameters type of the old and new constructor. These cases are studied individually to present more clear results.

4.3 Select DSUs

One of the important concerns in this study is the choice of the DSU tools to run the candidate patterns. According to the following rules, DSUs have been selected: i) The target tools should support dynamic updates in Java 7. Some DSUs support earlier versions of Java that cannot be applicable to our study [37]. ii) They have reasonable flexibility to support different changes. This allows us to run the maximum number of samples. In Kim's proposed DSU [76] and also UpgradeJ [23] only adding fields and methods to the original classes are permitted. Deleting or modifying operations are not allowed. iii)

In order to provide equal conditions for running samples, the dynamic update process should be transparent to developers. In this case, all samples can be executed in DSUs without any modification to support update process. iv) They should have the ability to determine the start time of the update process. However, we have to make some modification on two selected DSUs to add this feature. v) Finally, and most importantly, the DSU tool should be available. Some DSUs are no longer alive. Based on the mentioned points, three DSUs have been selected: JRebel, JavAdaptor, and DCE VM. Here is a brief explanation of these tools. More complete descriptions are given in Chapter 5.

JRebel is the only commercial product[2] that provides a quite flexible class reloading on the application level without JVM modification. It has been built with the capabilities developed for Javeleon [52]. Although the lack of support for changing the hierarchy of the type has been mentioned in the literature [54], our experiments indicate that this feature has been added. However, there is a lack of a mechanism for state transforming between two versions. JRebel has been developed for *'Edit and Continue'* purpose. It can be executed as a plug-in or standalone. JRebel provides a plug-in for the three main Java IDEs: Eclipse, IntelliJ, and NetBeans. It includes an agent that can be run alongside the program. This agent monitors the running application's class folder and applies any changes dynamically. To perform the update at a particular point, we just replace the modified classes by calling a method in the old code. The agent can detect these changes and perform the dynamic update immediately. All the patterns are executed through JRebel 7.0.3 on Java 1.7.0_79.

Dynamic Code Evolution VM (DCE VM) follows a different approach for class reloading than the previous tool and operates on JVM level instead of on top of it. It extends the operation of current HotSwap to provide dynamic update ability for the running program. The code evolution step in DCE VM is triggered by the *Java Debug Wire Protocol* (JDWP) [5]. The patched JVM allows the developer to add/remove methods/fields to the classes as well as modify inheritance hierarchy. Although the tool has been implemented for a certain JVM, the patch can be applied to other JVMs. The source code and binary are available at `http://ssw.jku.at/dcevm`. Switching from one version to another can easily be performed by calling a method.

Like JRebel, **JavAdaptor** has been built on top of the HotSwap. It supports internal class changes as long as the updated program remains type-safe. It also allows external class changes by rewriting bytecode thanks to some techniques such as containers and proxies. Moreover, JavAdaptor exploits one-to-one

[2] `https://zeroturnaround.com/software/jrebel/`

mappings for each field of the old class that has its counterpart in the new class. It also automatically initializes newly added fields with default values. Unlike two previous tools, the swinging execution phenomenon does not appear in the performing of all patterns. Because JavAdaptor keeps old loaded classes in JVM and they are still accessible after the update. This issue is referred as a *binary-incompatible update* [50]. To avoid this situation, the authors proposed a mechanism to invalidate the removed methods, constructors, and field accessors by rewriting their body immediately after the update. However, in some changes, the old code is still used after the update and the sample does not participate in the swinging execution. We will mark these cases in the results. JavAdaptor basically is developed as an Eclipse plug-in. While the application is running, the developer makes arbitrary changes and applies a new version of the classes by pressing a button inside the IDE.

4.4 Experimental Results

After specifying the parameters and choosing the DSUs to perform error-prone patterns, we execute all of the samples one-by-one and put the results in the Table 4.2. For ease of reference to the table rows inside the text, each row is marked with a number. These numbers are in line with Table 4.1. Each number includes two parts separated by a dash sign. The first part represents the row number in Table 4.1 and the second part represents the column number in Table 4.1. The rows and columns numbers are also marked in Table 4.1. The second column represents the change item and the third column indicates the type of change that has been applied to the item. Due to the merging some columns in Table 4.1 as well as the various results obtained from a change in an item, some rows have the same numbers. But anyway, the type of item and the change description are clear. The remaining columns are related to the parameters information on each DSU. The sign ✔means existence and sign ✖means the absence. Sign ━means no information. In the following, the results are described separately for each parameter.

			JRebel			DCE VM			JavAdaptor			
numbers from Table 4.1	*item*	*change type*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*annotate by static analyzer*
1-2	class	remove before first use	✗	✓	✗	✗	✗	✓	✗	✓	✗	✓
1-2	class	remove after first use	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓
1-5	class	change visibility	✓	✗	✓	✗	✓	✗	✗	✗	✓	✓
1-7	class	add final	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗
1-7	class	remove final	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗
1-8	class	add abstract	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓
1-8	class	remove abstract	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗
1-11	class	change implemented interfaces	✓	✗	✓	✗	✓	✗	✓	–	–	✓
1-12	class	change inheritance	✓	✗	✓	✗	✓	✗	✓	–	–	✓
2-2	interface	remove	✗	✗	✓	✗	✓	✗	✗	✗	✓	✓
2-5	interface	change visibility	✓	✗	✓	✗	✓	✗	✗	✗	✓	✓
2-12	interface	change inheritance	✓	✗	✓	✗	✓	✗	✓	–	–	✓
3-6	inner class	add static	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓
3-6	inner class	remove static	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓
3-13	inner class	move to the sub/super class	✗	✗	✓	✓	✓	✗	✗	✗	✓	✓
3-14	inner class	change nesting	✗	✗	✓	✓	✓	✗	✗	✗	✓	✓
4-2	method in class	remove	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓
4-3	method in class	change type with implicit conversion	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗
4-3	method in class	change type without implicit conversion	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓
4-5	method in class	change visibility	✓	✗	✓	✓	✓	✗	✓	–	–	✓
4-6	method in class	add static	✓	✗	✗	✓	✗	✗	✓	–	–	✗
4-6	method in class	remove static	✓	✓	✗	✓	✓	✗	✓	–	–	✓
4-7	method in class	add final	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
4-7	method in class	remove final	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓
4-8	method in class	add abstract	✓	✗	✗	✓	✗	✗	✓	–	–	✓
4-8	method in class	remove abstract	✓	✗	✗	✓	✗	✗	✓	–	–	✓

			JRebel			DCE VM			JavAdaptor			
numbers from Table 4.1	*item*	*change type*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*annotate by static analyzer*
4-9	method in class	add synchronized	✔	✖	✖	✔	✖	✖	✔	–	–	✖
4-9	method in class	remove synchronized	✔	✖	✖	✔	✖	✖	✔	–	–	✖
4-9	method in class	deadlock through synchronized	✔	✖	✔	✔	✖	✔	✔	–	–	✖
4-13	method in class	move to the sub/supper class	✔	✖	✖	✔	✖	✖	✔	–	–	✖
5-1	Constructor	add a constructor without parameter	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
5-1	Constructor	add a constructor with parameter	✔	✔	✖	✔	✔	✖	✔	✔	✖	✔
5-2	Constructor	remove the only existing constructor without parameter	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
5-2	Constructor	remove constructor with parameter	✔	✔	✖	✔	✔	✖	✔	✔	✖	✔
5-3	Constructor	change type with implicit conversion	✔	✔	✖	✔	✔	✖	✔	✔	✖	✖
5-3	Constructor	change type without implicit conversion	✔	✔	✖	✔	✔	✖	✔	✔	✖	✔
5-5	Constructor	change visibility	✔	✖	✔	✔	✔	✖	✔	–	–	✔
6-2	method in interface	remove	✔	✖	✔	✖	✔	✖	✔	✖	✔	✔
6-3	method in interface	change type with implicit conversion	✔	✔	✖	✔	✔	✖	✔	✔	✖	✖
6-3	method in interface	change type without implicit conversion	✔	✔	✖	✔	✔	✖	✔	✔	✖	✔
6-13	method in interface	move to the supper class	✔	✖	✖	✔	✖	✖	✔	–	–	✖
6-13	method in interface	move to the sub class	✔	✖	✖	✔	✖	✖	✔	–	–	✔
7-2	field in class	remove	✔	✖	✔	✔	✔	✖	✔	✔	✖	✔
7-3	field in class	change type with implicit conversion	✔	✖	✔	✔	✔	✖	✔	✔	✖	✖
7-3	field in class	change type without implicit conversion	✔	✖	✔	✔	✔	✖	✔	✔	✖	✔

			JRebel			DCE VM			JavAdaptor			
numbers from Table 4.1	*item*	*change type*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*annotate by static analyzer*
7-4	field in class	change value of a field	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
7-4	field in class	change value of a static final field	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
7-5	field in class	change visibility	✔	✖	✔	✔	✔	✖	✔	–	–	✔
7-6	field in class	add static	✔	✖	✖	✖	✔	✖	✔	–	–	✖
7-6	field in class	remove static	✔	✖	✔	✖	✔	✖	✔	–	–	✔
7-7	field in class	add final	✔	✖	✔	✔	✔	✖	✖	✖	✔	✔
7-7	field in class	remove final	✔	✖	✖	✔	✖	✖	✖	✖	✖	✖
7-10	field in class	add transient	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
7-10	field in class	remove transient	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
7-10	field in class	add volatile	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
7-10	field in class	remove volatile	✔	✖	✖	✔	✖	✖	✔	✖	✖	✖
7-13	field in class	move to a supper class	✔	✖	✖	✔	✖	✖	✔	–	–	✖
8-2	field in interface	remove a field	✔	✖	✔	✔	✖	✔	✔	–	–	✔
8-3	field in interface	change type with implicit conversion	✔	✖	✔	✔	✖	✔	✔	–	–	✖
8-3	field in interface	change type without implicit conversion	✔	✖	✔	✔	✖	✔	✔	–	–	✔
8-4	field in interface	change value of field	✖	✖	✔	✖	✖	✔	✔	–	–	✔
8-13	field in interface	move to a supper class	✔	✖	✖	✔	✖	✖	✔	–	–	✖
9-2	enum	remove an enum before first use	✖	✔	✖	✖	✖	✔	✖	✔	✖	✔
9-2	enum	remove an enum after first use	✖	✖	✔	✖	✖	✔	✖	✖	✔	✔

			JRebel			DCE VM			JavAdaptor			
numbers from Table 4.1	*item*	*change type*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*supported by DSU*	*runtime error*	*syntax error*	*annotate by static analyzer*
9-5	enum	change visibility	✔	✖	✔	✖	✔	✖	✖	✖	✔	✔
10-2	enum	remove a value	✔	✖	✔	✖	✔	✖	✔	✔	✖	✔
10-3	enum	change a type with implicit conversion	✔	✔	✖	✖	✔	✖	✔	–	–	✖
10-3	enum	change a type without implicit conversion	✔	✔	✖	✖	✔	✖	✔	–	–	✔
10-4	enum	change a value	✔	✖	✖	✖	✔	✖	✖	✖	✔	✖
12-2	annotation	remove annotation	✖	✔	✖	✖	✔	✖	✖	✖	✔	✔
12-2	annotation	remove an item annotation	✔	✔	✖	✔	✔	✖	✔	–	–	✔
12-3	annotation	change an annotation item type with implicit conversion	✔	✔	✖	✔	✔	✖	✔	–	–	✖
12-3	annotation	change an annotation item type without implicit conversion	✔	✔	✖	✔	✔	✖	✔	–	–	✔
12-4	annotation	change an annotation item value	✔	✖	✖	✔	✖	✖	✔	–	–	✖
12-12	annotation	change inheritance of an annotation	✔	✔	✖	✔	✔	✖	✔	–	–	✔

Table 4.2: *Experimental results related to the execution of error-prone patterns.*

4.4.1 Support by DSU

The results of this step are shown in the first column of each DSUs section in Table 4.2. In most cases, the changes are supported by the DSUs. Supported changes are determined by ✔and unsupported changes are specified by ✖number. It is obvious that JRebel is more flexible for supporting the changes. There are some cases that need to be explained in the table that includes the following:

As previously mentioned, JRebel detects the modified class files by monitoring the running application's class folder. Although it works well for modified classes, JRebel's agent does not recognize removed files. So, as you can see in the Table 4.2, removing a class (1-2), an interface (2-2), an enum (9-2), and an annotation (12-2) cannot be detected by JRebel. The similar thing happens for (3-13) and (3-14). When an inner class moves to the sub/super class or its nesting is changed, the old

inner class file is deleted and a new class file is added to the application. These cases can not be detected by JRebel.

In DCE VM, when a class is removed from an application (1-2), any access to the old class is still valid. In fact, when the current version of a class is not found, the previous version is used. A similar thing happens for removing an interface (2-2), an enum (9-2), and an annotation (12-2). However, for the removed interface, the program is finished by throwing the `UnsupportedOperationException` exception. Since it does not support the changing of class's implemented interfaces, when an interface is removed from an application, the classes which are implemented that interface are changed. Unlike JRebel, DCE VM accepts inner class hierarchy (3-13) and nesting (3-14) changes.

All selected DSUs support the change of the value of a final static field in a class. It is expected that similar situation should exist for the field on an interface. Because interface fields implicitly have final static modifiers. However, the change of the value of a field in an interface is not supported by the DSUs.

DCE VM does not allow any changes in class modifiers except *abstract*. Moreover, inheritance and implemented interfaces cannot be changed. The same situation applies to the interfaces. Also, any changes in Enum is not allowed in DCE VM. Finally, when a static modifier is added/removed from a field (7-6), the updated program produces `IncompatibleClassChangeError`.

JavAdaptor makes various copies of class files to use them in the updating process. When a class is deleted from the program files, it remains in other folders and old files are reused. Even if the file is removed from all folders, a runtime error occurs in the JavAdaptor. So, we identified the removal of a class (1-2), an interface (2-2), enum (9-2), and annotation (12-2) as unsupported changes. As mentioned in JRebel, the same thing happens in (13-13) and (3-14). When an inner class moves in a sub/super class or its nesting changes, the old inner class file is deleted and a new class file is added to the application.

JavAdaptor removes the final modifier from classes, methods, and fields before loading classes. Therefore, adding/removing the final modifier has no effect and we marked adding/removing final modifier as an unsupported modification (1-7,4-7,7-7). A similar case happens for the visibility. JavAdaptor changes the visibility of classes and interfaces to the public and makes the change of this modifier ineffective. Finally, changing the enum values (10-4) is not supported by the JavAdaptor.

4.4.2 Runtime Error in Dynamic Update Process

The results of this step are indicated in the second column of each DSU in the Table 4.2. ✖shows absence of error, ✔demonstrates the existence of error and ━indicates that the sample does not participate in the swinging execution.

A part of the runtime errors in DCE VM is related to the fact that these changes are not supported by DSU tool. Therefore, all cases in the table that have been

marked as unsupported, make a runtime error. However, for deleted classes (1-2) and enums (9-2) it does not create a runtime error because, as mentioned earlier, when a class is removed from an application and DCE VM does not find the new version, the previous version of the class is used and it does not create a runtime error. The types of exceptions are different: `UnsupportedOperationException`, `IncompatibleClassChangeError`, `VerifyError`, and `NullPointerException`. For JRebel and JavAdaptor, in the case of removed class (1-2) and removed enum (9-2), if they are not used before the update, the program encounters `NoClassDefFoundError`. Other templates that are not supported by these two tools do not create a runtime error.

The errors that are not related to the unsupported changes are described below. First, the cases that are same in all DSUs are expressed:

When a class is changed to an abstract class (1-8), it should not be instantiated. This change is supported by all DSUs, but after the update, a time error is triggered. The template of this change is written in the way that the altered class is instantiated in the old code after the update. In JRebel and DCE VM, it makes `InstantiationException` and in JavAdaptor, it produces `NullPointerException`.

When a static modifier is added to an inner class (3-6), and the old code attempts to create an object from that class after updating, the program stops by throwing the `NoSuchMethodError` exception. This situation happens in all of the DSUs. Considering the exception type, it can be concluded that the old code attempts to create an object from static class by calling the constructor that no longer exists.

In (4-2), the error is clear. When a method is removed from a class, any attempt to call it from the old code after the update makes a `NoSuchMethodError` exception. The same situation exists for changing the type of parameters or return types of methods (4-3). Due to the overloading capability in Java, any change in the type of parameters or the type of return of a method is interpreted as the addition of a new method. Thus, any attempt to invoke the method with the previous signature will be broken. The class constructors have a similar situation in the case of parameter type changes.

In (4-6), when a static modifier is removed from a method, access to it through the class must not be valid. So, when the old code calls the method through the class, the method does not exist in the class definition and the program is broken by throwing a `NullPointerException` exception. Error tracking reveals that JRebel calls the method by Java reflection technique. Therefore, searching the method inside the class returns a Null pointer and calling a Null pointer produces mentioned error. However, DCE VM uses direct calling and the program is finished with this error: «java.lang.IncompatibleClassChangeError: Expecting non-static method... ». If this pattern is updated through JavAdaptor, it does not participate in swinging execution.

Two types of constructors can be added to a class (5-1): with and without parameters. Adding a constructor without parameters does not make a runtime error

at the update time. By creating an object in the old code, it is called implicitly. Instead, adding a constructor with parameters to the class definition causes a `NullPointerException` error. Similar to what was described for the method invoking, JRebel exploits Java reflection to call constructors. Adding a constructor with parameter invalidates the default constructor of the class. Thus, the updated program cannot find default constructor and tries to invoke a returned null pointer. Both DCE VM and JavAdaptor have a similar behavior and create `NoSuchMethodError` on adding a constructor with a parameter.

At the removing the constructor (5-2), if the class has only one constructor without parameter, removing this constructor cannot introduce a runtime error. Because the default constructor can call implicitly. But if the class has a constructor with parameters, removing it will cause a runtime error similar to what happens in a method removal. The exception is `NoSuchMethodError`.

When the return type or the type of a parameter of a method changes in an interface (6-3), the corresponding methods in the classes that implement this interface are changed. Therefore, this is similar to (4-3) and it causes the same runtime error.

When the type of constants of an enum is changed (10-3), the constructor and accessing methods are changed. These changes are similar to the (5-3) and (4-3). Thus, they make same error on JRebel. However, DCE VM does not allow any kind of changes in the enum and in the all of the cases at 9 and 10 rows in Table 4.1, it makes the same error: « java.lang.VerifyError: verifier detected internal inconsistency or security problem ». In JavAdaptor, the sample uses the old code after the update and swinging execution does not affect this pattern.

When an item is removed from an annotation (12-2), any access to this removed item through the old code after the update causes a runtime error. The exception type is `NoSuchMethodError` in JRebel. in DCE VM, the program is broken with `annotation.AnnotationTypeMismatchException` exception. The same error happens when the item type is changed in the annotation (12-3). In JavAdaptor, for all of the cases related to the annotation, patterns use the old code after the update and there is no runtime error.

By using `@Inherited` in the definition of annotation, subclasses inherit this annotation from the superclass. If this part of the annotation definition is removed (12-12), any attempt to read this annotation in subclasses will fail and the program will end with the `NullPointerException` exception. This happens when JRebel is used. If the program is updated by DCE VM, the error will be the `annotation.-AnnotationTypeMismatchException`.

In the following, we will give the description of runtime errors that are only relevant to the DCE VM. Employing DCE VM to perform a dynamic update, changing the visibility of class items, produces an `IllegalAccessError` exception. These items include the method (4-5), constructor(5-5), and field(7-5) of a class. We reduce the visibility of these items in the new version. This causes the runtime error because the old code tries to have an illegal access to the changed item.

```
public class  SuperClass {
  final void foo(){...}
}
public class  SubClass extends  SuperClass {
  void bar(){
    new  SubClass().foo();
    doUpdate();
    new  SubClass().foo();
  }
}
```

(a) *old code.*

```
public class  SuperClass {
  void foo(){...}
}
public class  SubClass extends  SuperClass {
  @Override
  void foo(){...}
  void bar(){
    new  SubClass().foo();
    //doUpdate();
    new  SubClass().foo();
  }
}
```

(b) *new code.*

Figure 4.3: *Pattern for removing the* `final` *modifier from a method.*

In (4-7), when the final modifier is removed from a method signature, the program is broken with a `NullPointerException` exception. This pattern is shown in Figure 4.3. As shown, `foo` is a final method in the old version and is not overridden in the subclass. However, in the new version, `foo` can be overridden. While the program is running in the method `bar`, an update is performed and after that, calling `foo` through the subclass object causes a runtime error.

In (7-2), when a field is removed from a class, accessing to this field from the old code after the update may cause `NoSuchFieldError` exception. A similar error occurs when the type of a field is changed (7-3). Because when the type of a field changes in the new version, it is assumed that the old field has been deleted from the class and a new field with the modified type has been added to the class. So, it is logical that the same error appears.

Finally, in (7-7), when a final modifier is added to a field, any attempt to change its value from the old code after the update will put the program in an illegal mode by producing the `IllegalAccessError` exception. JRebel and JavAdaptor have no additional runtime errors to explain in this section.

4.4.3 Syntactic Errors in Dynamic Update Process

Execution of some patterns does not have a runtime error, but the program violates some programming language rules during or after the update. This may lead to unusual behaviors by the program. In this subsection, we review these states, which are named *syntax errors*. We describe only those cases that are supported by DSUs and do not have a runtime error. As previously mentioned, with the existence of a runtime error in a pattern, investigating the syntax error is meaningless. The results of this step are shown in the Table 4.2 in the third column of each DSU. Firstly, we explain the results which are shared between DSUs.

When a class or enum is removed from a program (1-2,9-2), it is still available after its update. As mentioned in the previous section, deleting a class or an enum can create a runtime error in JRebel and JavAdaptor if the class or the enum is not used before removing. However, if they have already been loaded, deleting the class file cannot affect the execution of the program and they are still available. This is not logical. The same thing happens when an interface is removed (2-2) in JRebel and JavAdaptor. But in DCE VM, this change causes runtime errors due to unsupported changes in the classes that implement the deleted interface.

As previously mentioned, changing the value of a static final field (7-4) is supported by DSUs and there is no runtime or syntax error. A similar situation is expected on changing the value of a field of the interface (8-4) because the field of the interface implicitly is a static final type. However, DSUs do not update the value of fields in the interfaces, and the old values of the fields are used after the update. This can create unexpected behavior in the program.

The rest of this section is intended to describe the syntax errors that occur when using JRebel. However, many of the following cases are not supported by the DCE VM or create a runtime error. Also, JavAdaptor does not participate in the swinging execution in some of these patterns.

When an interface is removed from the signature of a class (1-11), any assignment of the object of this class to a type of removed interface should not be valid after the update. This is a syntax error that occurs in JRebel. A similar situation happens when the inheritance of a class (1-12) or an interface (2-12) is changed. While there is an implicit type casting between an object of a subclass and the superclass type in the old version, this assignment should not be valid after the update. But JRebel permits to keep it. JavAdaptor does not participate in the swinging execution in these patterns.

When a method is removed from an interface (6-2), all classes that implement this interface may not implement the removed method in the new version. Two states happen: First, the method is removed from the class. This is equal to the case (2-4) and produces a runtime error. Second, this method is preserved in the class. In this case, if there is a type of interface in the old code, calling the removed method through the type of interface should not be valid. However, JRebel and JavAdaptor

violate this rule and mentioned invoking is still valid after the update.

In the change of visibility, JRebel does not respect the visibility reduction and continues the previous privacy policy after the update. For instance, suppose that the visibility of the method reduces from public to private. Calling this method out of the class should not be valid after the update. While JRebel executes the code of new version of the method, it adheres to the visibility of the previous version. This is a clear breach of privacy. We got the same results for reducing visibility in a class (1-5), an interface (2-5), a method in a class (4-5), a constructor (5-5), a field in a class (7-5), and an enum (9-5).

Another notable point here is the change in the synchronized modifier of a method (4-9). As shown in the List 4.2, a particular sequence of changes in the synchronized modifier of the methods encounters the running program with a deadlock situation. Although the old and new versions of the program can be run individually without any deadlock, a special change in the synchronized modifier of methods, as well as updates at a specific point, can be stuck the program in a deadlock.

When a field is removed from a class (7-2) or an interface (8-2), the field is still accessible from the old code after the update. The removed field is also accessible for new objects. A similar situation occurs to change the type of a field in a class (7-3) or in an interface (8-3). As explained earlier, when the type of a field is changed, it is interpreted that the field is removed and a new field from another type is added. However, the old field is accessible.

The non-static field is not accessible through a class. When a static modifier is removed from a field definition (7-6), access to the field after the update is an illegal operation. However, JRebel ignores this fact and continues to access the field through the class after updating.

Another important syntax error occurs in JRebel when a final modifier is added to the definition of a field (7-7). Normally, the value of a final field must not be changed during the program execution. However, in the JRebel, when a final modifier is added to the field definition, the value of this field can be changed after the update. As previously mentioned, JavAdaptor eliminates the final modifiers from fields definitions before loading classes. Finally, the removed enum value(10-2) is accessible from the old code after the update.

4.5 Apply to the Proposed Framework

After running the patterns on the three different DSUs and investigating the results, we found a large number of cases that the program finishes with producing an exception because of choosing an unsafe point to start the update process. In addition, we discover many cases that are not logical or violate programming language rules. To determine the role of the proposed validation framework in identifying runtime and syntax errors, all patterns are analyzed with the static analyzer of the framework.

Among the 75 samples, 44 ones have been annotated as unsafe. The last column of Table 4.2 shows these results. Then the patterns are performed with regard the annotated parts of the code. In the patterns that contain unsafe executable, the update process is postponed until finishing the execution of the unsafe method. This process covers most runtime and logical errors. However, as shown in the Table 4.2, there are some patterns that are not covered by the static analyzer (false negative) as well as some other patterns that include annotated methods but the updating process is not failed without regard to unsafe points (false positive).

	all patterns	*detected by validator*	*unsupported by DSU*	*supported by DSU*	*runtime error from supported*	*detected by validator*	*without runtime error*	*syntax error*	*detected by validator*	*without any error*	*detected by validator*
JRebel	75	44	9	66	19	14	47	20	17	27	4
DCE VM	75	44	22	53	27	22	26	4	2	22	3
JavAdaptor	75	44	18	57	16	12	41	1	1	40	0
Outcome	75	44	6	47	30	24	24	20	17	20	0

Table 4.3: *Summarize the results of code smell experiments.*

For a better understanding, we present the statistical summary results in Table 4.3. Each row is related to a DSU tool and the last row indicates the outcome of the results. The first column represents the number of patterns that have been examined. This number is 75 in all rows. The second column shows the total number of annotated samples. This value is the same for all rows (44). The next two columns represent the number of changes supported/unsupported by the DSUs. As shown, JRebel has the maximum number of supported changes and the DCE VM has a minimum. The sum of these two columns is equal to the number of patterns (75). The last row in this column shows the number of changes that are supported/unsupported by all DSUs. The fifth column indicates the number of patterns that terminates with a runtime error when updating without using the validator. Obviously, they are counted from the supported samples. The last row in this column refers to the number of cases that end with a runtime error in at least one of the DSUs. As discussed in the previous section, we do not examine samples that are not supported by the DSU. The next column presents the number of error-prone patterns that can be detected by our static analyzer. As you can see, there is a difference between columns 5 and 6. This means that the static analyzer cannot detect all runtime errors. We traced

these differences from Table 4.2 and extracted the error patterns that can not be identified by the static analyst. These patterns include cases: 4-3, 5-3, 6-3, 10-3, and 12-3. These errors are critical and we will show in the next section how the static analyzer should be enhanced to cover these issues. The next column indicates the number of cases that have no runtime error at the time of dynamic update. It is clear that unsupported cases by DSUs are not counted in this column.

The results in column 8 refer to the number of syntax errors that occur at the update time. Patterns that are not supported by the DSUs, as well as those that have a runtime error, are not considered here. The number of syntax errors in JRebel is 20, whereas this number is only 1 for JavAdaptor. This big gap is due to the fact that many considered patterns do not participate in the swinging execution. These cases are not counted here. The last row in this column shows the maximum number of syntax errors that occur in all DSUs. The next column represents the number of patterns from the previous column that can be identified by the static analyzer. As described above for runtime errors, there is also the difference between columns 8 and 9. The static analyzer cannot detect all syntax errors. In spite of the fact that the importance of syntax errors is less than runtime errors, the enhanced static analyzer should also cover such cases.

Column 10 demonstrates the number of cases supported by the DSU and does not include any runtime or syntax error. It is expected that these items should not be annotated by the static analyzer, but as shown in the last column, some of them are marked as unsafe for starting the dynamic update process. These include three cases for both of JRebel and DCE VM: add/remove abstract modifier of a method (4-8) and move to subclass for a method in an interface (6-13). However, this number is zero for the JavAdaptor. These three false positives make the validator a bit conservative. Because at the time of update, the DSU should tolerate an unnecessary waiting time to pass these cases. However, compared with conservative policy, proposed validation framework is still very efficient. It should be noted that according to the results in Chapter 3, it is roughly 82% better than conservative politics.

It can be summarized that the study identified 30 code smells causing fatal errors. Also, 20 smells have been identified that make syntax errors. However, these are slightly different in various DSUs, as well as smells of runtime errors and smells of syntax errors overlapping. Finally, there is no claim that this collection is complete. But given the fact that all the features of the programming language and all possible changes are considered, the completeness of this set is almost clear.

4.6 Enhance Static Analyzer

As mentioned in the previous section, during the examination of the patterns, it was found that some cases that include runtime or syntax errors cannot be detected by

the static analyzer. So, by extracting and investigating these cases, we are trying to enhance the proposed static analyzer to cover this defect. The most important failure is related to the runtime errors. Some cases end with a runtime error, but the static analyzer cannot predict this situation. These cases are similar in various DSUs and are related to the type change. In fact, when there is an implicit conversion between old and new types, the static analyzer cannot recognize the error. In all the cases, the type of created error is similar: `NoSuchMethodError`. To find the reason, we carefully studied the automatic annotating process for the pattern of changing the type of method (4-3). The old and new versions of the program are shown in List 4.4. The type of `foo` parameter is changed from `int` to `long` in the new version. If the program is updated while it is running inside the method `bar`, it creates a runtime error. Because the program searches for the `foo` method with the `int` parameter, which is not exist after the update. As described in Section 3.3, for determining the unsafe points, the static analyzer generates a temporary code and checks this code with the bug-finder tool. The temporary generated code for this sample is shown in List 4.4c. Bug-finder tool cannot find any compiler error in this code. Because there is an implicit type conversion between `int` and `long`. The easiest way to solve this problem is to simulate the situation that may occur at the update time by removing the method `foo`. Therefore, the temporary code will have a compiler error and the method `bar` is annotated as an unsafe method. Thus, in order to properly detect this situation in the enhanced static analyzer, along with adding the old method, the corresponding changed item should be removed from the temporary code.

This change in the static analyzer covers all runtime errors that were not previously covered. In addition, with this change, two uncovered patterns that may cause syntax errors (7-3, 8-3), have been covered. The only syntax error that is not covered by the static analyzer is the deadlock situation that occurs due to the specific sequence of synchronized modifier on two methods (4-9). As mentioned earlier, the first step of automatic annotating is to extract two versions changes. The information obtained from this step is used to identify probable deadlock pattern at the update time.

4.7 Summary

In this chapter, we determine which change patterns can fail the update process. We developed 75 error-prone patterns that each pattern nominates an atomic change for program elements. These changes include class internal modifications such as a field's type change as well as external changes such as a change in inheritance. These error-prone patterns are investigated by three DSU systems to extract unsafe patterns. Then we exploit our static analyzer to determine which unsafe patterns can be detected by our static analyzer. We found some gaps in determining all unsafe patterns. So, the static analyzer is enhanced to cover this defect. Finally, we listed patterns with errors as code smells.

```
public class A {
  void foo(int x){ ... }
  void bar(){
    ...
    doUpdate();
    int x;
    foo(x);
    ...
  }
  ...
}
```

(a) *old code.*

```
public class A {
  void foo(long x){ ... }
  void bar(){
    ...
    //doUpdate();
    long x;
    foo(x);
    ...
  }
  ...
}
```

(b) *new code.*

```
public class A {
  void foo(long x){ ... }
  void bar(){
    ...
    doUpdate();
    int x;
    foo(x);
    ...
  }
  ...
}
```

(c) *temporary generated code.*

Figure 4.4: *Pattern for modifying the parameter type of a method.*

5
Related Works

As mentioned earlier, one of the important parameters in determining the correctness of an update is choosing the start point of the update process. Starting an update process usually means starting the swinging execution, which can lead to temporary instabilities and possibly the failure of the update process. At the beginning of this chapter, we will introduce different policies that various DSU systems follow for choosing the safe update points. We describe the advantages and disadvantages of each policy.

Moreover, as it was illustrated in section 2.5, there is no general solution for ensuring a valid update in DSU system. So, it should be considered in each system individually. At the lack of common solution, each DSU system follows a definite policy in this regard. Even the absence of an explicit policy means ignoring this issue and this is in some way a policy. In this chapter, we list some DSU systems that have been developed for Java. We briefly describe each system and explain its encounter with the validation issue. Furthermore, some of the works in the literature have not proposed a new DSU system, but they tried to address this problem individually. In the last part, we will also briefly explain these efforts.

5.1 Determining a Safe Update Point

Regardless of the techniques that DSU tools use in the update process, they follow a specific policy for determining the starting moment of the update. The first category includes systems that no constraint is respected to start the update process. These systems have been developed as an IDE's plug-in to *edit and continue* purpose. JavAdaptor [106], JRebel [72], and Javeleon [52] are placed in this category. These tools are used in the coding phase of an application development cycle and the programmer is free to stop and rerun a program continuously. The waiting time is more critical in this application of DSU. Although it is perceived that the program stops at worst situation and developer can start it again but as it is mentioned in this thesis, the logical errors may happen and this can confuse programmer.

The second category contains systems that they verify some constraints before starting the update. In Jvolve [117], the update can be applied if the deleted or modified methods are not active. The user can add some methods to this blacklist.

This issue is so conservative and may lead to endless waiting. If an always resident method on the call stack is changed in the new version, the DSU system waits for this method to leave the stack. It never happens and the update process never starts. In UpgradeJ [23] and the Kim's DSU system [76], only adding the new fields and methods are allowed. With these restrictions, the problem of the swinging execution is removed. However, this limitation is a major defect and not acceptable in a real application. The similar situation happens in Java HotSwap [37] that only the body of the methods can be changed.

The third policy is to set predefined update points inside the old code. For instance, Rubah [101] as a DSU system follows this policy. The developer should modify the old code and calls a method inside it. This method checks the existence of a new version of the program and applies the update at this point. There is no way to detect this point automatically. However, the writer believes that every long-running program has a main loop and it is highly recommended to put this method calling inside the main loop. In multithread systems, it is difficult that all of the threads reach this point at the same time and the program may be caught in a deadlock situation. Although, the writer proposes an algorithm to prevent this situation. Our approach also can be used in this policy. So, before starting the update process, the unsafe parts of the old code are determined by our static analyzer and if the predefined update point is inside an unsafe area, then the developer should decide to make a sub-update which only changes the predefined update point in the old code and puts it in a safe part of the code. This policy is also used in some of the tools that have been developed to support dynamic update in other languages like C [69, 93].

5.2 Validation in Java DSUs

In Section 2.3, dynamic update systems for Java programs were generally reviewed. According to the mechanism exploited to implement dynamic update, these systems are divided into two main categories: Modifying the JVM and rewriting the program code. In the first technique, JVM is customized to support dynamic update. It is more flexible, efficient, and easy to use. However, it should be implemented for every JVM individually and requires maintenance in the case of JVM's upgrade. In the second technique, instead of modifying the JVM, the target application code is rewritten to give it dynamically update capability. Usually, these systems exploit container and proxy techniques. This method is more portable but less flexible.

Regardless of the mechanism used by the DSU system, validation is one of the major challenges in all the systems. The remainder of this section describes the various dynamic update systems in Java and their approach to the validation issue.

JDrums

The Java Distributed Run-Time Update Management System (JDrums)[110] is an implementation of a dynamic Java VM based on JDK 1.2. JDrums as a JVM patch exploits an object table to add an extra level of indirection to the internal representation of objects in the JVM. This indirection helps JDrums to replace the outdated classes with the new version. JDrums also uses a conversion class to specify how to transform each class. The user can customize the conversion class. It supports all of the internal class changes but prohibits superclass modifications. The main limitation of this approach is that the Just-In-Time (JIT) compiler should be disabled. It lets the program to run in the interpreted mode and reduces performance significantly. Moreover, JDrums can not transform the state of each object's superclass.

JDrums follows the conservative policy to start updating process. It needs none of the methods from each modified classes are active at the update time. The update is not performed if any changed method is active. The application continues to run the old version until it satisfies the requirements. For the future works, authors plan to add updating active objects and replacing method that can be found on the call stack.

DVM

Malabarba et al. modify Oracle's HotSpot VM version 1.2 to present a dynamic evolution system called Dynamic classes-enabled Virtual Machine (DVM)[89]. They change JVM to add type safety checks upon patch loading and exploit a mark-and-sweep algorithm to convert old objects to the new version. DVM uses the interpreter and cannot handle code evolution in the context of just-in-time compilation. It makes a significant performance penalty in contrast to normal execution. However, their main performance loss comes from employing a global lock at the bytecodes which include at least a method call or an object reference.

All kind of changes are permitted on DVM but it needs that the updated program is type safe. DVM tries to make a valid update by imposing some restrictions on the update. Active running methods cannot be updated as well as class interfaces. Like other JVM based solution, DVM suffers from dependency on a particular machine.

HotSwap

HotSwapping is a capability that was added to the most JVMs since JDK 1.4 [8]. It was developed based on Dimitriev work[37]. Even HotSwapping is not a standard Java feature, it is implemented by all of the well-known JVMs such as HotSpot JVM[4], JRockit JVM [7], and IBM's JVM [1]. Each JVM may have own implementation of this feature. HotSwap allows the developer to replace method body dynamically but it does not support adding or removing of methods or fields as well as changing

the supertypes of a class. These limitations define this system as the least flexible among all DSU systems.

HotSwap, the simplest DSU system has the slightest restriction at the time of starting the update process. The new version of the program can be applied at an arbitrary point of program execution. After the replacing the new code of a method, if the old method is active on the stack, it executes the old code until the call returns. Future calls will use the new version. As it is mentioned before this swinging execution may lead the program to be crashed. Therefore, HotSwapping does not have any mechanism for validating the DSU process.

DUSC

Orso et al.[97] present a code based technique for updating Java programs dynamically. It allows substituting, adding, and removing classes without stopping the running program. This technique does not need any support from runtime environment and therefore can be applied to any running program on standard JVM. This technique has been implemented as a tool called DUSC (Dynamic Updating through Swapping of Classes). It rewrites the program code and creates four separate classes for each class: i) implementation class for preserving the methods and fields of the original class ii) interface class for switching between various versions of class iii) wrapper class for managing the inter-class communication iv) state class for transforming the state. DSUC performs an update by swapping classes dynamically at runtime.

DUSC is a bit more flexible than HotSwap. The new version can add fields and methods as well as redefine all fields and methods which are already defined in the initial version. It does not support class schema changes. Due to the use of the proxy technique, this imposes a performance penalty on the program. Similar to the JDrums and DVM, for starting an update process, DSUC needs modified methods to become quiescent. Despite this conservative policy, the system may be faced an error.

JVolve

JVolve[117] introduces dynamic updating at the level of the JVM. It is implemented on Jikes Research Virtual Machine (RVM)[6] which is an open testbed for prototyping virtual machine technologies. JVolve customizes garbage collector to reload the new version of already loaded classes. It also generates particular state transformer classes automatically. JVolve does not burden any overhead during a program's steady-state execution. The new version of the program can include any internal changes such as adding/removing methods and fields. However, it refuses modification in the class inheritance hierarchy.

JVolve tries to accomplish update at the GC safe-points. When the garbage collector freezes the application threads for cleaning up unreferenced objects, all

threads must be at a safe-point in their execution. JVolve exploits this opportunity to perform the update. However, it checks that the modified methods are not active. If an item is found, the update process is postponed until the next safe-point. The developer can prepare a blacklist of methods; JVolve also should check that these methods are not active at the update time. Nevertheless, there is no specific criterion for providing this list. After trying out for a certain time, the update process will be failed.

DCE VM

Würthinger et al. introduced Dynamic Code Evolution VM (DCE VM)[1][126, 124] which is a patch for Java HotSpot VM to support dynamic code evolution on Java. Although Java HotSpot VM allows the body of methods to be changed, DCE VM permits any changes on the class members definitions such as adding/removing field/method. In addition, it supports class hierarchy changes, i.e, changing the superclass and implemented interfaces. It employs a garbage collector to redefine the new classes. Class redefinition is performed as an atomic operation in JVM safe-point. Moreover, DCE VM scans the heap in order to find pointers to the old classes and updates them to point to the new version. A customized mark-and-compact garbage collector increases the instance sizes in the case of added fields.

Like other JVM based solution, DCE VM is implemented for a special VM and it can be a major disadvantage. Also, it does not support custom program state transformation. For the time of update, DCE VM starts update process immediately and all of the instances are converted to the new version instantly. However, at the time of redefinition, the already invoked methods are resident on the call stack and can not be affected by the update. They continue executing the old code and as it is mentioned before it may cause a program crash.

Later they make an extension for DCE VM and map Aspect Oriented Programming (AOP) [74] to their programming model and present SafeWare[125]. It is a dynamic AOP system that provides atomic update capability for VM. SafeWeave can be applied as an aspect in the context of dynamic class loading. Aspect weaving does not impose the peak performance of VM.

JavAdaptor

JavAdaptor[105, 106] introduces a dynamic update system for Java programs by rewriting the program code. It is assumed that the underlying JVM has the HotSwapping feature. JavAdaptor is fully flexible and supports any kind of changes in the class definition and schema. It loads the new version of classes by renaming them

[1] http://ssw.jku.at/dcevm/

and searches the codes which refer to the old classes and modify them to access the new version. The body of outdated methods is easily updated by HotSwapping.

The main problem here is related to the type incompatibility between old and new versions. It is known as a *version barrier*[111]. JavAdaptor exploits containers and proxies techniques to make a level of indirection and solves this problem. In the same manner, the field is accessible through getter and setter functions. JavAdaptor utilizes the *Java Platform Debugger Architecture (JPDA)*[3] to encounter outdated instances. JPDA provides some APIs to find all objects of a modified class in the heap as well as objects which refer to those objects.

JavAdaptor basically is developed as an Eclipse plug-in. This application is well-known as an *'Edit and Continue'*. While the program is running, the developer making arbitrary changes and applying the new version of classes by pushing a button inside IDE. In this application, the main issue that should be considered is the waiting time. When the developer makes certain changes and pushes the button, he/she expects to see the result of those changes on the running application in the shortest possible time.

While immediate update reduces the user's waiting time, suffers from the *binary-incompatible update* [50] problem. To solve this problem, instantly after the update, JavAdaptor invalidates outdated methods by rewriting their body and put a throw exception code inside the methods[104]. Despite the fact that this mechanism reveals the illegal access to the outdated code after the update but it may cause to stop the running application.

JRebel

JRebel[2][72] is the only commercial tool that provides dynamic class reloading for Java programs without altering the JVM. Like JavAdaptor, it is developed for *'Edit and Continue'* purpose. It is implemented as a plug-in for some major Java IDEs such as Eclipse, IntelliJ, and NetBeans. However, it can be run stand alone as an agent. This agent can monitor the classpath of the running application and detect probable class changes and apply them to the running program. Although the early versions of JRebel do not support the class hierarchy changes[73], the current version is fully flexible and support all of the changes.

JRebel constructs an implementation class from the original classes. All the methods in the original class are moved to the implementation class. Each method is reformed to pass receiver object as the first argument. Moreover, all of the methods become static. Other classes still refer to the original class. When an invocation occurs, the original method finds current implementation of the class through JRebel APIs and calls corresponded method.

JRebel introduces performance penalty due to indirection techniques which are

[2]https://zeroturnaround.com/software/jrebel/

used. Moreover, it does not support customize state transformer. JRebel disregards to update correctness. It instantly applies the new version as soon as it is ready. Like JavAdaptor, it does not update the stack outdated code and it may make a runtime error.

Kim

Kim et al.[76, 75] propose a code based solution to redeploy a Java program with a new code dynamically. It can be run on each JVM with HotSwapping feature. They exploit proxy pattern[46] to support the unrestricted changes to Java classes. Special proxy classes play the role of intermediary between referring and referred classes. To support arbitrary changes, they move new items to the helper classes. All of the items are accessible through generic invoke methods. They employ HotSwap to update the caller of the new methods to refer to the new version. Despite the utilization of proxy technique, the performance reduction is negligible. This is a major advantage over systems that use indirect access techniques. They avail `invokespecial` and `invokeinterface` bytecode instructions which are optimized by JIT compiler. Using bytecode optimized commands distinguishing between direct access time and proxy access time very insignificantly. However, the use of proxy technique limits the class hierarchy changes.

This approach confesses that determining a specific program execution point which is safe to perform a dynamic update needs an entire understanding of application semantic[77]. Therefore, specifying such a safe update point automatically is an exceedingly complex task. In this case, the programmer can identify it manually. Kim et al. allow the programmer to specify update information through a simple configuration file which includes Java class, method, the statement number and, a probable thread synchronization code. Alternatively, the programmer directly identifies the safe update point inside the code and insert synchronization code manually.

Javeleon

Gregersen et al.[53, 51] developed Javeleon with the aim of easy-to-use tool to support DSU on Java application. Javeleon tightly integrates with NetBeans platform to take advantage of benefits of integration with specific frameworks, component systems, and application servers. It provides state-preserving arbitrary runtime evolution involving changes on class definitions and hierarchy. Their solution is based on proxy technique without modifying the JVM or language extension.

The code execution component of Javeleon is a middleware which is used to delegate the request to the most recent version of classes. It accomplishes the *In-Place Proxification* technique in composition with appropriate correspondence handling. Javeleon needs to replace all of the program components even when a

small change occurs. Gregersen et al. claim just 15% performance penalty [54] but another work measures 80% performance overhead on running HyperSQL case study[104]. Basically, Javeleon is developed for giving immediate feedback to the developer during the developing process. Therefore, it applies the ready changes instantly.

Later, *zeroturnaround* bought Javeleon and named it as Gosh![56]. In addition, JRebel from the same company enhanced with the capabilities brought in by the technology developed for Javeleon[108].

Rubah

Rubah[101, 34] is another portable DSU system for Java that works on stock VMs. Rubah's updating model is inspired by the Kitsune[66], a dynamic updating system for C program. Rubah is quite flexible and permits arbitrary changes on classes except for Java runtime classes and libraries that cannot be updated dynamically. Updatable classes can refer non-updatable classes directly but not the inverse. However, practically library classes do not directly refer to the application classes. Rubah provides two algorithms for performing state transformation: one parallel algorithm that transfers entire state at once, and other one is a lazy algorithm that transfers state on demand. Nevertheless, the parallel transformation is slow for exploiting large heaps and performance is decreased in comparison to the steady-state performance. Rubah also employs a GC-style manner to find and transform updated instances.

Rubah does not provide a complete transparency for the programmers and they need to learn how to inject Rubah's code for the future update. The developer should define update points inside the program code that identify safe moments to perform updates. Rubah provides some APIs for this purpose. There are no certain criteria for determining a safe update point. However, it is assumed that long-running programs usually have a main loop and it is recommended that update points be inserted within this loop. In addition, the developer must add some codes to perform control flow migration. Finally, default update class may be required to customized by the programmer.

UpgradeJ

Bierman et al. [23] introduced UpgradeJ, which is an extension to the Java programming language with support for upgrading classes dynamically. UpgradeJ extends Java language syntactically and obliges classes to be marked with the version number. It permits multiple, co-existing versions of classes and provides dynamically upgrade from one version of a class to another. Figure 5.1 shows an example of UpgradeJ code. UpgradeJ introduces a number of novel features to Java-like programming languages: explicit versions of classes, fixed version and upgradeable version objects, an upgrade statement, new class, revision, and exact version types.

```
class Button[1] extends Widget[1] {
        Font[1] font = new Font[1=]();
        Colour[2] colour = new Colour[3+]();
}
```

Figure 5.1: *Example of UpgradeJ code.*

UpgradeJ allows co-existence of instances from different versions of a class with preserving type-safety. In this way, UpgradeJ can perform DSU without requiring whole program state transformation. Version checking can be optimized by the compiler. This extension does not impose any steady-state overhead to the running program. UpgradeJ is placed in a bunch of systems that needs the developer to determine program points where the update can happen. These approaches do not provide a solution to ensure that the identified points lead to a valid update. However, UpgradeJ introduces a strong type-system that avoids an immense category of incorrect updates. Finally, and most importantly, UpgradeJ is not implemented yet and the authors plan to produce a prototype base on Java.

5.2.1 Summary

Table 5.1 summarizes the DSU systems strategies for determining the safe update point. As can be understood, the DSU system policy for identifying the safe update point is independent of the mechanism. Even the overall categorization of these systems into two types (code rewriting and JVM modification) does not affect this policy. For instance, JVolve and DCE VM are developed as a patch of JVM but have a different approach in validation. JVolve is the only system that supports user's blacklist of unsafe methods.

We will continue this chapter by looking at the efforts that have been made on the validation issue, regardless of the DSU systems. However, some works are designed and evaluated based on a certain DSU system. Of course, they have tried to generalize their method to other systems.

5.3 DSU Validation Efforts

Tedsuto [100] has been developed to test a program before doing a dynamic update. The basic idea of this framework is running existing system tests many times and explore the program behaviors systematically when the update is applied at the different points during the test's execution. Although Tedsuto is implemented for a specific DSU system (Rubah), they argue that it is a general solution and applicable to other state-of-the-art DSU systems. Tedsuto tightly depends on human interaction in many aspects which is a time-consuming and error-prone process. In addition, Tedsuto supposes that the DSU system produces a small number of

DSU system	Immediately	No active method	User blacklist	Predefined point
JDrums		•		
DVM		•		
HotSwap	•			
DUSC		•		
JVolve		•	•	
DCEVM				•
JavAdaptor	•			
JRebel	•			
Kim				•
Javeleon	•			
Rubah				•
UpgradeJ				•

Table 5.1: *Safe update point determination policy on Java DSU systems.*

update opportunities per interaction. This may not be true for all DSU systems. For instance, running this method on H2-test suite generates an enormous number of update opportunities. Therefore, they did not perform exhaustive testing with it. Finally, 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.

Zhao et al. [128] performed an exploratory study to find a safe update point. They statically extract unchanged methods from the classes and initially mark all their lines of code as a candidate update points. Then they reduce 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 [128]. 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 also the new version of the application is consistent with its original specification. All constraints quite unrealistic in a real-world application.

Gregersen et al.[55] identified some runtime phenomena that are intrinsic to dynamic updated Java applications. They consider two issues: i) The impact of an application's design on the ability to be updated dynamically. ii) How the dynamic updating influences our perception of the correct behavior of an updated application. To achieve these goals, they initially introduce five phenomena. They explore these phenomena by investigating the dynamic evolution of a graphical game through Javeleon - their DSU tool. These phenomena include phantom objects, transient

inconsistency, oblivious update, broken assumption, and lost state. They show that the runtime behavior of an updated application depends on the application design. The specific design may result in a different runtime behavior that only is revealed on dynamically update not on the traditional halt, redeploy and restart scheme. They explain how the underlying design in many cases can improve the prevention of these phenomena. Application developers can remedy these phenomena effect by following best-practice guidelines. In spite of the fact that the recommended design guidelines may scare developers from exploiting a language-transparent dynamic updating system; they do not give the application 100% guarantee that unwanted phenomena do not occur after a dynamic update. In addition, they do not propose any automatic way to detect these phenomena.

Some works attempt to improve the correctness of the dynamic update process through automatically generating state transformation for predefined update points. Magill et al.[85] give manually selected update points and the test cases which can be passed by both old and new versions and automatically produce transformer functions for updated fields. *Targeted Object Synthesis* (TOS) processes old and new programs memory snapshots and extracts old and new objects of updated classes. Then, it analyzes objects to produce the state transformation function. However, TOS requires that the programmer sets the corresponding points in two versions. Later, Zhao et al.[127] enhanced TOS by filtering the candidate points and increasing the speed of the process. The success of these methods tightly depends on the selecting appropriate points. As it is presented in this work, choosing improper point may cause serious consequences.

Another work tries to enhance the reliability of dynamic update by employing recovery techniques. ADSU[58] is a DSU system that leverage *Automatic Runtime Recovery* (ARR) techniques[26] to recover runtime errors caused by improper dynamic updating. ADSU exploits lightweight ARR approach[84] that can handle errors caused by invalid memory access. They just discard invalid write and synthesizes a type-specific default value for invalid read. Even though this technique might be useful in the case that the default behavior is the desired behavior for updates, the semantics of the old objects may be lost after the recovery and thus ADSU is failed. They have postponed detecting the semantic relation between two versions of an object as a future work.

Although our focus is on Java-based DSU systems, some efforts have been performed in the other languages in this regard. For instance, Hayden et al.[63] propose a systematic testing to find the safe update points for C programs. They put the candidate update points before each method calling and test each update point with the program test cases. In this way, 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. Other work presents a methodology for automatically verifying the correctness of dynamic update[64]. The programmers can express the desired properties of an updated program using *client-*

oriented specifications (CO-specs). These properties can be verified automatically by using off-the-shelf tools. The quality of process depends on the user properly specified properties.

6 Conclusion

Although it has been formally proven that in general the validation of dynamic updating systems is undecidable, this issue can be considered in each program separately. In this dissertation, we showed that 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. DSU system should respect to some constraints before starting the update process. We provided some facilities for the developer to express update constraints as well as a runtime validator to verify these constraints to reach a safe update point. Moreover, we presented an automatic static analyzer to determine unsafe update points as constraints. In addition, we identified the code smells in the dynamic update; patterns that can create runtime or syntax errors in the dynamic update process. This goal is realized by developing a set of error-prone patterns in Java and examining them in state-of-the-art DSUs.

6.1 Contributions

The primary contributions of this dissertation are:

Meta-data. Each 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 as well as which constraints should be respected in order to proceed with the update. These constraints can be expressed in the code as a meta-data. We proposed a set of meta-data that can be exploited by the developer to explain the constraints. The static and dynamic constraints can be introduced by employing these meta-data. 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.

Validation process. The expressed constraints should be verified at the update time. To provide this service for DSUs, we presented an architecture of validation process to find a safe update point before starting the update process. It includes

a *validator* that can communicate with both the running program and DSU tool to find a safe update point by considering the specified constraints. This portable component can be employed by different DSUs as a pre-update part.

Swinging execution phenomenon. Although the proposed meta-data are useful in describing constraints, the annotating process is time-consuming and potentially error-prone when manually done. Moreover, since the code by definition is a continuous evolution, also the related annotations should be updated accordingly at each change. Therefore, we tried to find an automatic way to determine some constraints and decorate code automatically. So, we studied the execution model of a program during and after the dynamic update precisely and introduced the swinging execution phenomenon which can make a transient inconsistency on the program execution at the update time.

Static analyzer. Usually, each part of the program can participate in the swinging execution. However, some code participation may create a runtime error at the update time. We proposed an automatic way to determine which part of the code is unsafe to start the update process and participate in the swinging execution. This novel approach statically anticipates the swinging execution impact on each changed part of the code and determines unsafe codes. This method is implemented as a static analyzer that takes two versions source code and gives an annotated code. This unique tool is quite fast even for the big programs. Running this method on various versions of three long-running applications demonstrated almost 82% improvement in comparison with the conservative approach.

Error-prone patterns. We also tried to examine the impact of any small change on the validation of a DSU process. To achieve this goal, we designed and developed a set of candidate error-prone patterns. Each pattern nominates an atomic simple change that can occur in a program evolution. We develop more than 75 individual candidate error-prone patterns based on Java language features and possible changes for each item. These patterns are used to explore code smells on the dynamic software updating. In addition, we believe that this set can be exploited as a reference set by each DSU tool to measure flexibility. Also, developers can check the behavior of their DSU tool in dealing with different possible changes.

DSU code smells. Code smell or bad smell refers to any symptom in the program code that possibly indicates a deeper problem. For the first time, we introduced the code smells that may cause a run-time or syntax error on the dynamic update process. To explore these smells, we traced the dynamic evolution of candidate error-prone patterns on at least three state-of-the-art DSUs and categorized the results. We establish code smells according to the detected 30 runtime errors and 20 syntax errors with some overlaps in different DSUs.

Enhance the static analyzer. Finally, we examined the error-prone patterns by our static analyzer and understood that a few number of the patterns cannot be detected by the analyzer. So, we tracked these situations and enhanced the static analyzer to cover this defect and can disclose all of the code smells.

6.2 Future Work

In this work, we focused on the static analysis techniques to determine some constraints automatically. The analyzer can detect the unsafe part of the code based on predicting runtime and syntax errors. Although predicting these errors and dodging them at update time is a big step toward a valid update, this is not sufficient for ensuring a safe and correct update. Along with fatal errors, the program may have some wrong behaviors during or after the update. One of the upcoming tasks can be the prediction of this kind of misbehavior and prevent them.

Different DSUs have been developed for Java language which is one of the most popular object-oriented programming languages. However, as it has been mentioned in the related works, most of them suffer from lack of a validation mechanism. Almost there is a similar situation for the DSU systems in other programming languages. Even though the idea of this thesis has been implemented for Java DSUs, it can easily be extended to the DSUs in other languages as a future work.

Our approach is experimentally proved by evaluating in some real server applications. We have applied our method to various versions of different programs. Even we have implemented 75 patterns which nominate all atomic changes in a program evolution and examined them by our method. However, a formal approach can be helpful to prove the correctness of this method.

Some DSU systems have been implemented as a plug-in for well-known IDEs like Eclipse and NetBeans. The programmer can apply changes to the running application by pushing a button. Whereas our method can be performed automatically, it needs the user to run every step manually. Moreover, it is independent of any IDE and be executed from command line. To improve the usability of the framework, it can be implemented as a plug-in for IDEs and integrated with DSUs.

Bibliography

[1] developerworks ibm developer kits. http://www.ibm.com/developerworks/java/jdk/. Accessed September 2017.

[2] Eclipse categorized java problems. https://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/compiler/CategorizedProblem.html. Accessed September 2017.

[3] Java se 7 java platform debugger architecture (jpda). http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/. Accessed July 2017.

[4] Java se hotspot at a glance. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html. Accessed September 2017.

[5] Java(tm) debug wire protocol. http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html. Accessed September 2017.

[6] Jikes rvm. http://www.jikesrvm.org/. Accessed September 2017.

[7] Oracle jrockit. http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html. Accessed September 2017.

[8] Oracle(tm). java se 1.4 enhancements. http://download.java.net/jdk8/docs/technotes/guides/jpda/enhancements1.4.html. Accessed July 2017.

[9] Tiobe index. https://www.tiobe.com/tiobe-index/. Accessed July 2017.

[10] Write once, run anywhere? http://www.computerweekly.com/feature/Write-once-run-anywhere. Accessed July 2017.

[11] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: Online patches and updates for security. In *USENIX Security Symposium*, pages 287–302, 2005.

[12] Joe Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.

[13] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[14] Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In Doug Grant, editor, *Proceedings of the 13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75, Canberra, Australia, August 2001. IEEE.

[15] Lowell Jay Arthur. *Software evolution: the software maintenance challenge*. Wiley-Interscience, 1988.

[16] Malcolm P Atkinson, Francois Bancilhon, David J DeWitt, Klaus R Dittrich, David Maier, and Stanley B Zdonik. The object-oriented database system manifesto. In *DOOD*, volume 89, pages 40–57, 1989.

[17] Andrew Baumann. Dynamic update for operating systems. *Doctor of Philosophy, School of Computer Science and Engineering, The University of New South Wales*, 112, 2007.

[18] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W Wisniewski. Improving operating system availability with dynamic update. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pages 21–27, 2004.

[19] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *USENIX Annual Technical Conference, General Track*, pages 279–291, 2005.

[20] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing Dynamic Update in an Operating System. In Vivek Pai, editor, *Proceedings of the USENIX Annual Technical Conference (ATAC'05)*, pages 279–291, Anaheim, CA, USA, April 2005.

[21] David M Beazley. *Python essential reference*. Addison-Wesley Professional, 2009.

[22] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE)*, pages 1–17, 2003.

[23] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In Jan Vitek, editor, *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, Lecture Notes in Computer Science 5142, pages 235–259, Paphos, Cyprus, July 2008. Springer.

[24] Toby Bloom and Mark Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[25] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Acm Sigplan Notices*, volume 38, pages 403–417. ACM, 2003.

[26] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791. IEEE Press, 2013.

[27] Walter Cazzola, Ruzanna Chitcyan, Awais Rashid, and Albert Shaqiri. μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures*, 2017.

[28] Walter Cazzola and Mehdi Jalili. Dodging Unsafe Update Points in Java Dynamic Updating Systems. In Alexander Romanovsky and Elena Troubitsyna, editors, *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE'16)*, pages 332–341, Ottawa, Canada, 23rd-27th of October 2016. IEEE.

[29] Walter Cazzola and Edoardo Vacchi. @Java: Bringing a Richer Annotation Model to Java. *Computer Languages, Systems & Structures*, 40(1):2–18, April 2014.

[30] Gang Chen, Hai Jin, Deqing Zou, Zhenkai Liang, Bing Bing Zhou, and Hao Wang. A framework for practical dynamic software updating. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):941–950, 2016.

[31] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44. ACM, 2006.

[32] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In Wolfgang Emmerich and Gregg Rothermel, editors, *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 271–281, Minneapolis, MN, USA, May 2007. IEEE.

[33] Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.

[34] Luis Gabriel Ganchinho de Pina. *Practical Dynamic Software Updating*. PhD thesis, INSTITUTO SUPERIOR TECNICO, 2016.

[35] Antonella Di Stefano, Giuseppe Pappalardo, and Emiliano Tramontana. An infrastructure for runtime evolution of software systems. In *Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on*, volume 2, pages 1129–1135. IEEE, 2004.

[36] Mikhail Dmitriev. Safe class and data evolution in large and long-lived java [tm] applications. 2001.

[37] Mikhail Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In Vinny Cahill, Siobhán Clarke, Simon Dobson, and Robert Filman, editors, *Proceedings of the 1st Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE'01)*, pages 14–18, Tampa Bay, FL, USA, October 2001.

[38] Peter Ebraert, Yves Vandewoude, Theo D'Hont, and Yolande Berbers. Pitfalls in Unanticipated Dynamic Software Evolution. In Walter Cazzola, Shigeru Chiba, Gunter Saake, and Tom Tourwé, editors, *Proceedings of ECOOP'2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, pages 3–8, Glasgow, Scotland, July 2005.

[39] Len Erlikh. Leveraging legacy system dollars for e-business. *IT professional*, 2(3):17–23, 2000.

[40] Robert S. Fabry. How to Design a System in Which Modules Can Be Changed on the Fly. In Raymond T. Yeh and C. V. Ramamoorthy, editors, *Proceedings of the 2nd international conference on Software engineering (ICSE'76)*, pages 470–476, San Francisco, CA, USA, October 1976. IEEE.

[41] Jean-Rémi Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-Grained and Accurate Source Code Differencing. In Marsha Chechik and Paul Grünbacher, editors, *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, pages 313–324, Västerås, Sweden, September 2014. IEEE.

[42] Brent Fulgham and Isaac Gouy. The computer language benchmarks game, 2010. *URL http://shootout. alioth. debian. org*, 2012.

[43] Richard P Gabriel, Jon L White, and Daniel G Bobrow. Clos: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991.

[44] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, January-February 2009.

[45] E Gamma and K Beck. The junit test framework, 2000.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Pioneers and Their Contributions to Software Engineering*, pages 361–388. Springer, 2001.

[47] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Safe and automatic live update for operating systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 279–292. ACM, 2013.

[48] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[49] Adele J Goldberg. Smalltalk-80: the interactive programming environment. 1984.

[50] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. Java (tm) language specification (the 3rd edition), 2005.

[51] Allan R. Gregersen, Bo Nørregaard Jørgensen, Hadaytullah, and Kai Koskimies. Javeleon: An Integrated Platform for Dynamic Software Updating and Its Application in Self-* Systems. In *Proceedings of the 2012 Spring Congress on Engineering and Technology (S-CET'12)*, pages 1–9, Xian, China, May 2012. IEEE.

[52] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic Update of Java Applications — Balancing Change Flexibility vs Programming Transparency. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):81–112, March/April 2009.

[53] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic update of java applications—balancing change flexibility vs programming transparency. *Journal of Software: Evolution and Process*, 21(2):81–112, 2009.

[54] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. 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)*, pages 6–15, Szeged, Hungary, September 2011.

[55] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. 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*, pages 6–15. ACM, 2011.

[56] Allan Raundahl Gregersen, Michael Rasmussen, and Bo Nørregaard Jørgensen. Dynamic Software Updating with Gosh!—Current Status and the Road Ahead. In José Cordeiro, David Marca, and Marten van Sinderen, editors, *Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT'13)*, pages 220–226, Reykjavík, Iceland, July 2013.

[57] Rean Griffith and Gail Kaiser. A runtime adaptation framework for native c and bytecode applications. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 93–104. IEEE, 2006.

[58] Tianxiao Gu, Zelin Zhao, Xiaoxing Ma, Chang Xu, Chun Cao, and Jian Lü. Improving reliability of dynamic software updating using runtime recovery. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 257–264. IEEE, 2016.

[59] Deepak Gupta. *On-line software version change*. PhD thesis, PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, 1994.

[60] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for On-Line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[61] Jens Gustavsson, Tom Staijen, and Uwe Assmann. Runtime evolution as an aspect. 2004.

[62] L Guy Jr. Steele. common lisp: The language, 1990.

[63] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades (HotSWUp'09)*, Orlando, FL, USA, October 2009.

[64] Christopher M Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S Foster. Specifying and verifying the correctness of dynamic software updates. *VSTTE*, 12:278–293, 2012.

[65] Christopher M Hayden, Karla Saur, Michael Hicks, and Jeffrey S Foster. A study of dynamic software update quiescence for multithreaded programs. In *Hot Topics in Software Upgrades (HotSWUp), 2012 Fourth Workshop on*, pages 6–10. IEEE, 2012.

[66] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, General-Purpose Dynamic Software Updating for C. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, pages 249–264, Tucson, AZ, USA, October 2012. ACM.

[67] Michael Hicks and Scott Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, November 2005.

[68] David Hovemeyer and William Pugh. Finding Bugs Is Easy. *ACM Sigplan Notices*, 39(12):92–106, December 2004.

[69] Mehdi Jalili, Saeed Parsa, and Habib Seifzadeh. A Hybrid Model in Dynamic Software Updating for C. In *Proceedings of International Conference on Advanced Software Engineering and Its Applications (ASEA'09)*, Communications in Computer and Information Science 59, pages 151–159, Jeju Island, South Korea, December 2009. Springer.

[70] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[71] Evgueni Kabanov. Method and arrangement for re-loading a class, September 4 2012. US Patent 8,261,297.

[72] Jevgeni Kabanov. JRebel Tool Demo. *Electronic Notes Theoretical Computer Science*, 264(4):51–57, February 2011.

[73] Jevgeni Kabanov and Varmo Vene. A Thousand Years of Productivity: The JRebel Story. *Software: Practice and Experience*, 44(1):105–127, January 2014.

[74] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242, 1997.

[75] Dong Kwan Kim. *Applying dynamic software updates to computationally-intensive applications*. PhD thesis, 2009.

[76] Dong Kwan Kim and Eli Tilevich. Overcoming JVM HotSwap Constraints Via Binary Rewriting. In Tudor Dumitraş, Danny Dig, and Iulian Neamtiu, editors, *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades (HotSWUp'08)*, Nashville, TN, USA, October 2008. ACM.

[77] Dong Kwan Kim, Eli Tilevich, and Calvin J Ribbens. Dynamic software updates for parallel high-performance applications. *Concurrency and Computation: Practice and Experience*, 23(4):415–434, 2011.

[78] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-Programming with Rascal. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE'09)*, LNCS 6491, pages 222–289, Braga, Portugal, July 2009. Springer.

[79] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering*, 16(11):1293–1306, 1990.

[80] Insup Lee. Dymos: a dynamic modification system. 1983.

[81] MM Lehman. Towards a theory of software evolution-and its practical impact. 2000.

[82] Bennet P Lientz and E Burton Swanson. Software maintenance management: a study of the maintenance of computer applications software in 487 data processing organizations, 1980.

[83] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[84] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices*, volume 49, pages 227–238. ACM, 2014.

[85] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S McKinley. Automating object transformations for dynamic software updating. In *ACM SIGPLAN Notices*, volume 47, pages 265–280. ACM, 2012.

[86] Kristis Makris. *Whole-program dynamic software updating*. Arizona State University, 2009.

[87] Kristis Makris and Rida A. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In Geoffrey M. Voelker and Alec Wolman, editors, *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*, pages 31–44, San Diego, CA, USA, June 2009.

[88] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. *ACM SIGOPS Operating Systems Review*, 41(3):327–340, 2007.

[89] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J Fritz Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP*, volume 14, pages 337–361. Springer, 2000.

[90] Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais, and Jean-Marc Jézéquel. Unifying runtime adaptation and design evolution. In *Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on*, volume 1, pages 104–109. IEEE, 2009.

[91] Iulian Neamtiu and Michael Hicks. Safe and Timely Updates to Multi-Threaded Programs. In Amer Diwan, editor, *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation (PLDI'09)*, pages 13–24, Dublin, Ireland, June 2009.

[92] Iulian Neamtiu, Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *ACM SIGPLAN Notices*, volume 43, pages 37–49. ACM, 2008.

[93] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.

[94] Agnes C. Noubissi, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Hot Updates for Java Based Smart Cards. In Rida Bazzi, Michael Hicks, and Carlo Zaniolo, editors, *Proceedings of the 3rd Workshop on Hot Topics in Software Upgrades (HotSWUp'11)*, pages 168–173, Hannover, Germany, April 2011. IEEE.

[95] David Oppenheimer, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A Patterson, and Katherine Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Transactions on Computers*, 51(2):100–107, 2002.

[96] Manuel Oriol. *An approach to the dynamic evolution of software systems*. PhD thesis, University of Geneva, 2004.

[97] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 649–658, Montréal, Canada, October 2002. IEEE Press.

[98] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Software—Practice and Experience*, 2015.

[99] Mathias Payer, Boris Bluntschli, and Thomas R Gross. Dynsec: On-the-fly code rewriting and repair. In *HotSWUp*, 2013.

[100] Luís Pina and Michael Hicks. Tedsuto: A General Framework for Testing Dynamic Software Updates. In Lionel Briand and Sarfraz Khurshid, editors, *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16)*, Chicago, IL, USA, April 2016. IEEE.

[101] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a Stock JVM. In Todd Millstein, editor, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'14)*, pages 103–119, Portland, OR, USA, October 2014. ACM.

[102] Susanne Cech Previtali and Thomas R Gross. Dynamic updating of software systems based on aspects. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 83–92. IEEE, 2006.

[103] Mario Pukall. Object roles and runtime adaptation in java. In *RAM-SE*, pages 33–37, 2008.

[104] Mario Pukall. *JAVADAPTOR: unrestricted dynamic updates of Java applications*. PhD thesis, Magdeburg, Universität, Diss., 2012, 2012.

[105] Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 989–991, Waikiki, Honolulu, Hawaii, on 21st-28th of May 2011. IEEE.

[106] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schöter, and Gunter Saake. JavAdaptor — Flexible Runtime Updates of Java Applications. *Software—Practice and Experience*, 43(2):153–185, February 2013.

[107] Mario Pukall, Christian Kästner, and Gunter Saake. Towards unanticipated runtime adaptation of java applications. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 85–92. IEEE, 2008.

[108] Rein Raudjärv and Allan Raundahl Gregersen. Jrebel. android: runtime class- and resource reloading for android. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 741–744. IEEE Press, 2015.

[109] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *ECOOP*, volume 2374, pages 205–230. Springer, 2002.

[110] Tobias Ritzau and Jesper Andersson. Dynamic deployment of java applications. In *Java for Embedded Systems Workshop*, volume 1, page 21. London, 2000.

[111] Yoshiki Sato and Shigeru Chiba. Loosely-separated "sister" namespaces in java. In *European Conference on Object-Oriented Programming*, pages 49–70. Springer, 2005.

[112] Donna Scott. Assessing the costs of application downtime. *Gartner Group, May*, 1998.

[113] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE software*, 10(2):53–65, 1993.

[114] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A Survey of Dynamic Software Updating. *Journal of Software: Evolution and Process*, 25(5):535–568, May 2013.

[115] Junrong Shen, Xi Sun, Gang Huang, Wenpin Jiao, Yanchun Sun, and Hong Mei. Towards a unified formal model for supporting mechanisms of dynamic component update. *ACM SIGSOFT Software Engineering Notes*, 30(5):80–89, 2005.

[116] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 29(4), August 2007.

[117] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. In Amer Diwan, editor, *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation (PLDI'09)*, pages 1–12, Dublin, Ireland, June 2009.

[118] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann, 2014.

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

[120] David Thomas, Chad Fowler, Andrew Hunt, et al. *Programming Ruby: the pragmatic programmer's guide*, volume 2. Pragmatic Bookshelf, 2005.

[121] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.

[122] Bruce F Webster. Pitfalls of object oriented development. 1995.

[123] Niklaus Wirth. Modula: A language for modular multiprogramming. *Software: Practice and Experience*, 7(1):1–35, 1977.

[124] Thomas Würthinger. Dynamic code evolution for java.

[125] Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and atomic run-time code evolution for java and its application to dynamic aop. In *ACM SIGPLAN Notices*, volume 46, pages 825–844. ACM, 2011.

[126] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic Code Evolution for Java. In Andreas Krall and Hanspeter Mössenböck, editors, *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java (PPPJ'10)*, pages 10–19, Vienna, Austria, September 2010.

[127] Zelin Zhao, Tianxiao Gu, Xiaoxing Ma, Chang Xu, and Jian Lü. Cure: Automated patch generation for dynamic software update. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 249–256. IEEE, 2016.

[128] Zelin Zhao, Xiaoxing Ma, Chang Xu, and Wenhua Yang. Automated Recommandation of Dynamic Software Update Points: An Exploratory Study. In Minghui Zhou and Charles Zhang, editors, *Proceedings of the 6th Asia-Pacific Symposium on Internetware (INTERNETWARE'14)*, pages 136–144, Hong Kong, China, November 2014. ACM.

[129] Benjamin Zorn. Personal communication, based on experience with microsoft windows customers, 2005.