Rubah: Efficient, General-purpose Dynamic Software Updating for Java

Luís Pina[*]
INESC-ID / Instituto Superior Técnico
Lisbon, Portugal

Michael Hicks
University of Maryland
College Park, USA

Abstract

This paper presents Rubah, a new dynamic software updating (DSU) system for Java programs that works on stock JVMs. Rubah supports a large range of program changes (e.g., changes to the class hierarchy and updates to running methods), does not restrict important programming idioms (e.g., reflection), and, as shown by performance experiments using an updatable version of the H2 database management system, imposes low overhead on normal execution.

1 Introduction

Dynamic software updating (DSU) frameworks are used to update programs while they run. Of particular interest to us are DSU frameworks that support nearly arbitrary program changes: in addition to changes to method code (as commonly supported by Java and .NET VMs) we would like to add and remove methods and fields, change their types, and even change a class's superclass and implemented interfaces. Such support is needed to facilitate dynamic updates corresponding to full releases, rather than simple (e.g., security) patches.

In the past, such highly flexible DSU services have been implemented in two ways: as a bytecode-to-bytecode transformation, or via an enhanced JVM. The appeal of the former approach is that it is JVM-independent, and so DSU services are not restricted to a particular platform. Most existing transformations work by introducing one or more proxy objects per application object whereby a dynamic update redirects a proxy to point to the new object version [9, 10]. This approach has several significant drawbacks. First, it can break application semantics and therefore restricts certain useful programming idioms; e.g., uses of reflection could reveal the presence of proxy objects and are thus forbidden. Second, and perhaps more importantly, proxy objects can add significant overhead to normal execution, e.g., up to 50%. This overhead is too high for practical use. The alternative of using a specialized VM, like the JVolve VM [11], can avoid such performance overheads and semantic restrictions (though Jvolve in particular does not support some dynamic updates, e.g., class hierarchy alterations). But a specialized VM limits adoption.

In this work-in-progress paper we present Rubah, a new transformation-based DSU framework for Java. Our goal in building Rubah is to retain the portability benefits of transformation-based techniques while avoiding their high overhead and semantic restrictions. We have done this by avoiding the use of proxy classes altogether: when a new dynamic update becomes available, we dynamically load the new versions of added or changed classes, and then perform what amounts to a full garbage collection (GC) of the program to locate and update all instances of affected classes. This approach is inspired by Jvolve, which employs a modified version of the VM's actual GC. To avoid changing the VM, Rubah instead introduces an application-level GC-style traversal implemented using reflection and some class-level rewriting. Rubah also borrows ideas from the recent Kitsune DSU system for C programs [6] to support updates to actively running methods (e.g., those running infinite loops) by requiring the programmer to make some simple changes to use the Rubah API.

We have implemented Rubah and evaluated it on H2, an SQL DBMS written in Java. So far we have implemented two full updates to H2. Using the TPC-C benchmark from the DaCapo benchmark suite [1], we confirm that the benchmark completes correctly even when H2 is updated midstream, and that Rubah adds about 8% overhead to normal execution, far less than prior transformation-based approaches. On the other hand, the pause that Rubah induces while the transformation takes place is currently prohibitively high, on the order

[*]This work was performed while this author was visiting the University of Maryland.

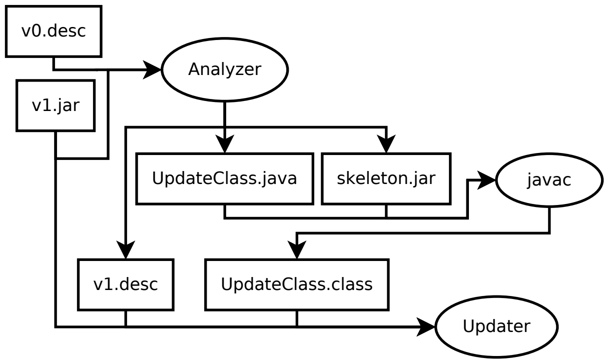

Figure 1: Preparing and installing an update using Rubah. Square boxes represent artifacts: Compiled code (jar/class), source code (java), or update descriptors (desc). Round boxes represent tools: Rubah's analyzer and updater, and the unmodified Java compiler (javac).

of 10 seconds. This is due to the time it takes perform HotSwap to introduce new code (about 2 seconds) and to traverse the whole heap using regular Java code. Nevertheless, our evaluation shows that Rubah is still faster than the alternative approach of saving H2's state, restarting the database server, and reading the state in the new version.

Implementing a DSU system as a specialized JVM or as a bytecode transformation and runtime library has always been considered fundamental, with the former benefiting flexibility and performance, and the latter benefiting platform independence. Ultimately, based on what we have learned so far, we believe the best answer might be to make small, palatable changes to the JVM upon which one can mostly implement DSU outside the JVM. We view Rubah as bounding what is possible without changing the JVM, and therefore highlighting what JVM support is ultimately needed. Rubah supports extremely flexible dynamic updates to Java programs imposing little run-time overhead and requiring few code changes. On the other hand, pause times are quite high, and runtime overhead could be lower still. We are currently exploring minimal changes to the JVM and/or its programming API that might be used to solve these problems. Identifying exactly how much support is required from the JVM is exciting future work.

In the remainder of the paper we present Rubah's approach in detail, discuss our implementation, and present our experience and performance evaluation using H2. We conclude with plans and ideas for our continuing research.

2 Rubah

Rubah implements *whole program updates* in the style of a previously developed DSU system for C, called Kitsune [6].[1] An updatable program starts in Rubah's *driver*, a bit of stub code that calls into the application's `main` method. The program then executes normally until an update becomes available. Rubah stops each thread when it reaches the first *update point* following the update becoming ready. Update points, implemented as calls to the method `Rubah.update`, can be placed at the head of long-running loops to ensure both safety and availability [7].

Once all threads are stopped this way, Rubah performs the *data migration*, which is a GC-style traversal of the heap that updates those objects whose classes have changed between versions. Next, Rubah performs the *control flow migration*: It restarts each thread t until t reaches the same update point in which t stopped. We refer the reader to the Kitsune paper [6] for more illustration of how this works; space constraints preclude further elaboration here.

In Rubah, nearly every class can be changed in a nearly arbitrary manner. The only *non-updatable classes* include the Java runtime classes and libraries (e.g., Java collections). Updatable classes can directly reference non-updatable classes but not the reverse, due to issues involving the bootstrap class path of a Java application [8]. Of course, library classes do not directly reference application classes, so this restriction poses no practical difficulty.

In the remainder of this section we focus on the two central novelties of Rubah: the update class, which specifies how updated objects are to be transformed, and Rubah's GC-style custom data migration.

The update class specifies how to migrate the program state from version `v0` to version `v1`.[2] Figure 2 shows an example of an update class. It has *instance conversion methods* that take an existing instance `o0` belonging to version `v0` and use it to initialize the equivalent new instance `o1` that shall replace `o0`. Update classes may also have *static conversion methods*.

The arguments of the conversion method in Figure 2 are *skeleton classes*. As the name implies, they are a skeleton of the classes defined in versions `v0` and `v1` with all fields and methods made public. Via the skeleton classes the programmer can refer to version `v0` or `v1` unambiguously and still use the regular Java compiler to

[1] Kitsune is the Japanese word for *fox*, a shape shifter. Rubah is the Indonesian word for fox; natives of the island of Java speak Indonesian.

[2] Throughout the remainder of this paper, we shall consider that updates take an application from version *v*0 to version *v*1 for the sake of clarity.

```
1  class UpdateClass {
2    void convert(
3        v0.org.h2.store.PageStore o0,
4        v1.org.h2.store.PageStore o1) {
5      o1.readCount = 0L;
6      o1.writeCount = 0L;
7      o1.writeCountBase = o0.writeCount;
8    }
9  }
```

Figure 2: Example of an update class with a single instance conversion method. Field `writeCount` in version `v0` was renamed to `writeCountBase`. Besides, version `v1` introduced two fields, `readCount` and `writeCount`, to keep how many bytes were read/written since the store was opened. The analyzer tool generates all code except for line 6, which the programmer added, and line 7, which originally contained `o1.writeCountBase = 0L`.

compile the update class.

Rubah generates the source code for most of the update class. As shown in Figure 1, the programmer uses Rubah's *analyzer* tool to generate a stub update class, along with the code that is loaded as part of dynamic update. The analyzer tool takes as input the version-`v1` compiled code and the *version descriptor* generated for version `v0`. The version descriptor contains metadata about a given version, such as a list of all updatable classes. Then, the analyzer tool generates the update descriptor for version `v1` along with the initial update class. The analyzer tool compares versions `v0` and `v1` and matches fields by owner class name, field name, and field type. It generates conversion methods for all classes with unmatched/changed fields and puts a default initialization of those fields inside each generated method. The programmer customizes the stub update class to migrate each unmatched/changed field according to version `v0`'s state. Once this is done, the operator provides the update class and the prepared `v1` code to the *updater* tool, which initiates the dynamic update process.

Summing up all the effort required to update an application using Rubah, the developer has to: (1) Specify update points, (2) write the control flow migration in each new version, and (3) specify the program state migration in the update class. Even though Rubah does not provide any tools for the programmer to assess the correctness of any the required manual interventions, it is our experience that tasks 1 and 2 can be easily verified during the development of the new version because an error introduced in this tasks typically terminates the program abruptly during the update process, even for trivial updates easy to simulate during development. Besides, both our previous experience with Kitsune [6] and the experimental evaluation we describe in Section 3 suggest that the effort required for task 2 takes place mostly in the first version. Maintaining this code in subsequent updates is usually trivial and requires little effort, if any.

```
1  Rubah.traverse(Object o0) {
2    if (converted[o0]) return;
3    Class c0 = o0.getClass();
4    Class c1 = Rubah.mapClass(c0);
5    Object o1 = Unsafe.allocate(c1);
6    converted[o0] = o1;
7    Rubah.copyUnchanged(o0, o1);
8    Rubah.convert(o0, o1);
9    o1.traverse();
10 }
```

Figure 3: Migration algorithm (simplified). For all loaded classes Rubah injects `$traverse` methods, which call method `Rubah.traverse` for each field. Line 8 calls the appropriate convert method specified in the update class.

We decided not to automate task 1 because previous work [7] shows that the only approach that results in 100% safe updates is manual specification of update points. As for task 3, Rubah automates it as much as possible to avoid programmer error. Nevertheless, previous work [5] explores how to verify the correctness of state migration; we leave that as future work.

Implementing data migration The updater signals the update to the application, whose driver uses a custom class loader to rewrite all loaded classes. When installing an update, the driver generates a new Java class for each changed class in the running application. This way, Rubah supports unrestricted modifications to updatable classes without requiring special support from the JVM. To avoid name collisions, Rubah renames each version of each updatable class. Rubah also replaces reflection calls by equivalent Rubah API calls that take the name mangling process into account. [3]

Rubah can traverse all program state without requiring special support from the JVM. To do so, it adds a method named `$traverse` to all classes, even non-updatable classes. This method, in turn, calls Rubah's API on all reference type fields of each object. Therefore, Rubah just needs to call the `$traverse` methods on root references (stopped threads and static fields of loaded classes) to find and migrate all updatable objects that the program keeps. Rubah has a tool to add the `$traverse` method to all the classes defined in the bootstrap jar, which is shipped with the JVM and contains all JDK classes. Then, the updatable program can be loaded as long as the JVM uses the processed jar as its bootstrap class path.

[3] Native methods in general, and reflection calls via native methods in particular, are not handled.

Rubah does not migrate local variables. However our experience from Rubah and Kitsune is that relevant state is always accessible from root references. As for external resources, such as open connections or files, Rubah provides API for the programmer to keep them open during the update process. It is up to the developer to migrate the control flow to avoid re-opening existing resources while updating. When evaluating Rubah, we used this mechanism to keep open connections established with an unmodified H2 client while an update takes place.

Figure 3 shows a simplified version of the migration algorithm. Rubah starts by checking if a given instance has been converted already. If not, it creates an instance `o1` of the updated class without running any constructor, registers `o0` as migrated to `o1`, copies all unchanged fields, runs the conversion method present in the update class, and finally traverses `o1`. At this point, Rubah has already replaced all references made by the update class to the skeleton types by references to the real types. Rubah runs this algorithm sequentially and provides no guarantee about the order in which it traverses the heap. When Rubah migrates all instances, it uses HotSwap [3] to update the code on classes that the update modifies but that do not require migration.

Rubah calls instance conversion methods in a similar way to how Java calls constructors [4]. Let us consider the case in which classes `A` and `B` are updatable, class `B` extends `A`, class `N` is non-updatable, and class `A` extends `N`. In this case, to convert instances of class `B`, Rubah: (1) copies all fields inherited from class `N`, (2) copies all unchanged fields from class `A`, (3) calls `A`'s conversion method to migrate `A`'s updated fields, (4) copies all unchanged fields from class `B`, and (5) calls `B`'s conversion method to migrate `B`'s updated fields.

For the migration algorithm to work, each field with an updatable type must be able to refer to multiple class versions, i.e., class `c0` or `c1`. As there is no subtyping relationship between the two classes, Rubah erases all updatable types from fields and method signatures, replacing updatable types by `java.lang.Object` and inserting casts to the right type on the code that uses those fields or methods.

When migrating the program state, Rubah must ensure that the identity of each new object `o1` is the same as the identity of the migrated object `o0`, even though `o1` may even belong to a different class. In Java, object identity is related with three concepts: (1) The `equals` method, (2) the == operator, and (3) the `hashCode` method. The heap traversal that Rubah performs deals with concepts 1 and 2 because it replaces all references to `o0` by a reference to a respective `o1`. For the hash code, Rubah adds a field named `$hashCode` to updatable classes, injects code at the beginning of contructors to initialize this field, and overrides the `hashCode` method (when not present) to return the value of this field. When migrating the program state, Rubah copies the value of this field from `o0` to `o1`.

Version	**LOC**	**Update Class**
1.2.121	36882 / 428 - 267 / 9	-
1.2.122	37182 / 428 - +0 / +0	106 / 12 - 45 / +2
1.2.123	37084 / 428 - +0 / +0	40 / 4 - 30 / +0

Figure 4: Programmer effort to support Rubah. The columns have the format 1/2-3/4. For column LOC, 1 and 2 are the total lines of code of the H2 version, excluding comments and blank lines, and the total number of files; 3 and 4 are the number of modified lines and files to support Rubah. For column Update Class, 1 and 2 are the number of lines of code and conversion methods generated automatically; 3 and 4 are the number of lines and methods that the programmer needs to change.

3 Evaluation

This section presents the details and results of the experimental evaluation that we conducted on Rubah. To evaluate Rubah, we quantify how much the programmer needs to rewrite the original program, how much overhead Rubah adds in steady state execution after the bytecode transformation, and how long is the program paused for Rubah to complete a dynamic update.

To evaluate Rubah we used H2, an SQL DBMS written in Java. We adapted the TPC-C benchmark used by the DaCapo benchmark suite [1] to connect to a server H2 process, instead of using H2 embedded in the benchmark process itself, so that we can update the server dynamically while an unmodified client keeps running. Besides profiling H2's performance, the TPC-C benchmark also checks the H2 database for semantic errors. This way, TPC-C confirms that Rubah transforms H2's bytecode and migrates its state correctly.

Programmer Effort. We used Rubah to update H2 from v1.2.121 to v1.2.122 and then to v1.2.123. The patches are available at `http://web.ist.utl.pt/luis.pina/hotswup13`. Figure 4 shows the changes we made to support DSU with Rubah. All required changes are made to version 1.2.121, and these are limited to just a few lines of a few files. We added two conversion methods to update v1.2.122 (beyond those generated automatically): One method converts a field that changed semantics (the values of integer constants changed) and another method forces H2 to reload changed property files before the update finishes.

Performance. We measured TPC-C throughput on each scenario over 11 benchmark runs, each one consists

	TTC (ms)	**Heap** (KB)
Original	102174 ± 1655	692733 ± 183720
Rubah	110443 ± 1519	800643 ± 192561
	TTU (ms)	**Heap** (KB)
v0v0	52319 ± 664	1089291 ± 402314
121 to 122	10190 ± 714	804539 ± 197813
122 to 123	10978 ± 1066	787484 ± 186687
	121 to 122	**122 to 123**
Export (ms)	3154.72 ± 331.87	3214.18 ± 235.82
Import (ms)	8249.72 ± 220.45	8275.72 ± 311.89
Total (ms)	13037.72 ± 626.68	13431.27 ± 621.49

Figure 5: Experimental evaluation of Rubah. This table reports the average of the measured values and the standard deviation. The TTC column reports the time it takes to complete a single iteration; TTU reports the time it takes to perform an update; Heap reports the heap used throughout each iteration. The export/import/total rows measure the time the old version takes to dump the database contents to an SQL file, the time the new version takes to import that SQL file, and the overall time since the dump starts to the import finishes including restarting the H2 server, respectively.

of 10 benchmark iterations, each one executing 32,000 transactions. We restarted the server and client processes between benchmark runs. To benchmark Rubah's memory usage, we separately take measurements by sending signal SIGQUIT every second to the server JVM, which causes it to dump overall heap usage.

All benchmarks ran on an Intel Core i5 750 processor (4 cores) with 12GB RAM, running a 64-bit Ubuntu GNU/Linux 12.10 with Oracle JVM version 1.6.0_41 (Runtime Environment build 1.6.0_41-b02, HotSpot 64-bit Server VM build 20.14-b01). The JVM maximum heap size is 3GB on the server and 1GB on the client.

Figure 5 reports the results. We can see that Rubah introduces about 8% overhead on the original application performance. As for memory overhead, we can see an increase of about 15% in the amount of used heap. The high standard deviation in the times is due to the JVM trying to reduce time spent in GC rather than reduce the used memory, given that the used heap size is much smaller than the maximum heap size.

The middle part of Figure 5 reports the time and heap needed to perform the update. Row v0v0 measures the cost of updating v0 to itself, "converting" every single instance. Real updates take less time because fewer objects must be transformed (though the entire heap is still traversed). The slow update times derive from the traversal making many normal method invocations per object; we are investigating means to optimize the traversal.

Without Rubah, the typical update scenario for H2 would be to export the database contents to disk, stop the server running in the old version, start the server in the new version, and import the database contents. We need to export/import the database state because this particular benchmark uses in-memory tables.

Given Rubah's slow update times, we performed an experiment to measure how long this update scenario would take. This experiment runs the benchmark once to populate the database with some data. Then, it exports/imports the database state, restarting the H2 server in between. The bottom part of Figure 5 reports the results measured over 11 executions of this experiment. Rubah is able to perform the update about 20% faster.

Even though 20% is just a small improvement, Rubah is a much better approach because it stops the application in a safe state and keeps all connections alive during the update process. Using Rubah, we did not change the clients in any way to survive the update process. Without Rubah, either the update process would disconnect all clients or the clients would require custom modifications to survive the update process (e.g. attempting to reconnect). Nevertheless, the Rubah pause time is still quite high, and ongoing work aims to address it.

4 Related Work

As mentioned in Section 2, Rubah employs the same approach to whole program updates as Kitsune [6], a DSU system for C; in particular, both systems employ the concepts of control and data migration, and update points. The main difference is that Rubah implements data migration via code inserted during bytecode transformation, and specifies updates via its update class. Kitsune's data migration algorithm is more brittle because of C's lack of type safety. Kitsune uses a domain-specific language to specify state conversions; Rubah's update class is a more compact, and natural, representation for conversions in the Java context.

Rubah's update class bears some similarity to PJama's *bulk conversion* [2] routines. However, these routines works on offline updates of persistent object stores, rather than on-line updates to running Java programs. PJama also does not use skeleton classes to refer to old/new state unambiguously.

There have been several prior systems that support DSU for Java without requiring VM support. JRebel [13] allows unrestricted changes to the structure of a class (add/remove fields/methods) but not to the class position in the hierarchy, which Rubah supports. JRebel also does not support any program state migration mechanism besides the default Java initialization to added fields. DUSC [9] and DUSTM [10] work by inserting proxies as an indirection to every object, and paying the respective steady-state performance penalty, which can be as high as 50% for a similar H2 benchmark.

The JVM itself is a natural place to support DSU. The Oracle JVM supports dynamic updates to method bodies in existing classes [3], for the purposes of enabling "stop-edit-continue" development. (JRebel also targets this domain.) Full-featured DSU is supported by the JVolve [11] and DCE VMs [12], though even these do not support some changes to the class hierarchy, which Rubah does. With DSU services are located inside the JVM itself, these systems can take advantage of internal mechanisms, such as the garbage collection and JIT compiling, to implement efficient support for DSU. However, this approach is inherently non-portable. The goal of building Rubah was to show that similarly powerful mechanisms can be built outside the VM while imposing comparable performance and development costs. As stated in the introduction, a hybrid approach may ultimately be the best answer.

5 Conclusion

This paper has presented our work to date on Rubah, a dynamic software updating system for Java that works on stock VMs. Rubah's applies whole-program updates to running programs, employing a novel means of specifying state changes (the update class) and a novel application-level, GC-style traversal to find and transform updated instances. Rubah's imposes relatively low overhead on normal execution, but more work is needed to reduce update pause times. Nevertheless, our experimental evaluation shows that Rubah's update times are faster than the typical technique of saving the state, restarting the application in the new version, and importing the saved state.

Future Work The most pressing problem of Rubah is the high update pause time. We are planning to explore *lazy* algorithms for transforming the state, to both amortize the cost of data migration and improve the update pause time. In the case that we are not able to do so, we are planning on identifying the minimum support that we require from the JVM for implementing low update pause times.

We also are working on expanding our experimental evaluation to more applications. Another possible way to expand Rubah would be researching tools that verify the correctness of update classes.

Acknowledgments. This work was partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia - under projects PTDC/EIA-EIA/108240/2008 and PEst-OE/EEI/LA0021/2011, and by NSF grant CCF-0910530.

References

[1] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 169–190.

[2] Dimitriev, M., and Atkinson, M. P. Evolutionary data conversion in the PJama persistent language. In *Proceedings of the Workshop on Object-Oriented Technology* (London, UK, 1999), Springer-Verlag, pp. 25–36.

[3] Dmitriev, M. Towards flexible and safe technology for runtime evolution of Java language applications. In *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference* (2001).

[4] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[5] Hayden, C. M., Magill, S., Hicks, M., Foster, N., and Foster, J. S. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)* (Jan. 2012), pp. 278–293.

[6] Hayden, C. M., Smith, E. K., Denchev, M., Hicks, M., and Foster, J. S. Kitsune: efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2012), OOPSLA '12, ACM, pp. 249–264.

[7] Hayden, C. M., Smith, E. K., Hardisty, E. A., Hicks, M., and Foster, J. S. Evaluating dynamic software update safety using efficient systematic testing. *IEEE Transactions on Software Engineering 38*, 6 (Dec. 2012), 1340–1354. Accepted September 2011.

[8] Liang, S., and Bracha, G. Dynamic class loading in the Java(TM) virtual machine. In *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices* (1998), ACM Press, pp. 36–44.

[9] Orso, A., Rao, A., and Harrold, M. A technique for dynamic updating of java software. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 649–.

[10] Pina, L., and Cachopo, J. DuSTM - dynamic software upgrades using software transactional memory. Tech. Rep. 32/2011, INESC-ID Lisboa, June 2011. Available at `http://www.inesc-id.pt/ficheiros/publicacoes/7298.pdf`.

[11] Subramanian, S., Hicks, M., and McKinley, K. S. Dynamic software updates: a VM-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 1–12.

[12] Würthinger, T., Wimmer, C., and Stadler, L. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java* (New York, NY, USA, 2010), PPPJ '10, ACM, pp. 10–19.

[13] ZeroTurnAround. JavaRebel. `http://www.zeroturnaround.com/jrebel/`.