Dynamic Software Updating with Gosh!
Current Status and the Road Ahead

Allan Raundahl Gregersen[1], Michael Rasmussen[1] and Bo Nørregaard Jørgensen[2]
[1] *ZeroTurnaround, Tartu, Estonia*
[2] *The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense, Denmark*

Keywords: Dynamic Software Updating, Unanticipated Run-time Evolution, Run-time Phenomena.

Abstract: Any non-trivial software system has to be upgraded regularly to incorporate bug fixes and security patches or simply to keep up with the inevitable evolution in end-user requirements. Software upgrading is challenging, especially when it comes to online upgrading of running systems. In this paper, we present the current status of Gosh!, a dynamic-software-updating system for Java, which provides comprehensive support for changing class definitions of live objects, including adding, removing and moving fields, methods, classes and interfaces anywhere in the inheritance hierarchy. Prior to the acquisition by zeroturnaround.com, Gosh! was known as Javeleon. In this paper we demonstrate the capabilities of Gosh! by performing a dynamic updating experiment on five consecutive revisions of the classical arcade game Breakout. Based on the result of this experiment we show that dynamic updating of class definitions for live objects may under some circumstances result in different run-time behavior than would be observed after a cold restart of the upgraded application. Finally, we conclude by discussing the implication of this finding for future research directions within dynamic software updating.

1 INTRODUCTION

Software is subject to continuous change, not only as part of its development cycles, but also over time to stay useful to its users (Lehman, 1997). In most standard deployment environments this implies that use of the next software version typically requires halting the currently running version before deploying and starting the new version. Using a dynamic software-updating system (DSU) this is no longer necessary, as the DSU system will dynamically replace the running version with the new version. Depending on how advanced the DSU system is this may happen more or less transparent to end-users. We say that a DSU system is end-user transparent if it does not require any intervention of end-users during an update, and similarly we say it is developer transparent if it does not require developers to take specific precautions. Hence, the two forms of transparency is a key quality for any DSU system, since it strongly influences the degree to which it will be successful. The success of DSU systems is especially important as software systems tend to become more complex in terms of internal run-time state and interactions with external systems. This trend is for instance present in mission-critical systems such as surveillance and control of air traffic, ground transportation, oil and gas production, industrial process, power generation, and smart-grids. These application domains are all subject to safety, environmental and economical regulations and restrictions, which make system downtime due to maintenance tasks like software updates not only inconvenient but also very expensive.

Where past research has contributed significantly toward making DSU practical for systems written in C or C++, upgrading of server functionality (Neamtiu et al., 2006; Chen et al., 2007; Makris and Bazzi, 2009), deploying security patches (Altekar et al., 2005), and operating systems upgrades (Soules et al., 2003; Baumann et al., 2005; Baumann et al., 2007; Makris and Ryu, 2007; Chen et al., 2006; Arnold and Kaashoek, 2009), there used to be a gap when it comes to systems written in managed languages, such as Java, Ruby, and C#. In the past DSU for managed languages was limited to HotSpot JVM (Sun Microsystems, 2004) for Java. For some .NET languages (Microsoft Corporation, 2008) a similar limited support of on-the-fly updating of

Raundahl Gregersen A., Rasmussen M. and Nørregaard Jørgensen B..
Dynamic Software Updating with Gosh! - Current Status and the Road Ahead.
DOI: 10.5220/0004562302200226
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 220-226
ISBN: 978-989-8565-68-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

method bodies applies. However, this support is too restricting for all but the simplest updates. Limiting changes to method bodies would render the DSU system useless for updating most of the revision improvements reported for the Jetty webserver in (Gustavson, 2003). Academic approaches (Ritzau and Andersson, 2000; Malabarba et al, 2000; Orso et al., 2002; Bierman et al., 2008) offer more flexibility, but remain still to be proven on realistic development scenarios. Furthermore, these approaches employ designs for method and object indirection, which impose substantial space and time overheads on steady-state execution. The lack of approaches supporting managed languages had the potential to become a severe problem as an increasing number of enterprise systems and embedded systems are written in those languages. Fortunately, the research on DSU for managed languages has caught up and includes now multiple promising approaches. State-of-the-art approaches for Java includes; JRebel (Kabanov, 2010), an application-level system which is currently the de-facto commercial tool for class reloading in Java; Dynamic Code Evolution VM (Würthinger et al., 2010), a VM-enhancement of the Java HotSwap™ VM (Dmitriev, 2001); JVolve (Subramanian et al., 2009), a VM approach based on the Jikes Research VM, and Gosh! (Gregersen and Jørgensen, 2009), an application-level system.

In this paper, we first provide an overview of code changes supported by DSU systems targeting Java; we then give an introduction to the design and implementation of Gosh!, followed by the latest development in the performance benchmarking of Gosh!. Then, we demonstrate the capabilities of Gosh! by applying it to a series of consecutive revisions of an in-house implementation of the classical arcade game Breakout. Finally, we discuss the result of this experiment and its implication for future research direction within dynamic software updating.

2 GOSH! COMPARED TO OTHER APPROACHES

A comparison of the code changes supported by DSU systems that are currently public available is given in table 1. As the table shows, Gosh! is at the moment the DSU system with the most comprehensive support for redefinition of Java classes. The Issues symbol in table 1 indicates that there are circumstances where the code change is not fully supported by the DSU system.

Tabel 1: DSU system comparison.

Code change	*Gosh!*	*JRebel*	*DCEVM*
Changes to method bodies	✓	✓	✓
Adding/removing fields	✓	✓	✓
Adding/removing methods	✓	✓	✓
Adding/removing constructors	✓	✓	✓
Adding/removing classes[i]	✓	✓	⚠
Replace superclass	✓	✗	✓
Adding/removing implemented interfaces	✓	✗	✓
Automatic new instance field initialization (developer-defined default value)[ii]	⚠	✗	✗
Automatic new static field initialization (developer-defined default value)[iii, iv]	⚠	⚠	✗
Move field to super class (preserving the state)[iv]	✓	✗	⚠
Move field to sub class (preserving the state)[iv]	✓	✗	⚠
Changing static field value[iii]	✗	⚠	✗
Changing primitive static final field value[v]	✓	⚠	⚠
Adding/removing enum values[vi]	✓	⚠	✗

✓ Supported ✗ Not supported ⚠ Issues

i. Only Gosh! and JRebel provide integration with custom class-loaders for adding new classes that is not present on the class path.
ii. Gosh! supports automatic field initialization without re-executing the constructor/static initializer. However, automatic initialization does currently not support branching (try/catch, ternary operator etc.)
iii. Currently, JRebel is the only approach with support for changing static field values. However, it is based on re-executing the entire static initializer which may lead to serious side-effects caused by repeated execution of code which should only execute once.
iv. Gosh! is currently the only DSU system cable of correctly transferring values of fields which have been moved up or down in the inheritance hierarchy. DCEVM copies field values to super/sub classes even in situations where the field is also retained in the former class version.
v. Only Gosh! fully supports changing primitive static final field values, as both JRebel and DCEVM gives wrong results for constant values accessed through reflection after updating.
vi. JRebel claims support. However, simple tests show that e.g. removing and adding enum values is not handled correctly in switch statements.

3 GOSH! DESIGN AND IMPLEMENTATION

The core idea of the Gosh! updating model is to allow multiple versions of the same objects to co-exist in a running system. This is achieved by creating new class loaders for each new version, thus setting up distinct type namespaces. Since this

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

approach imposes a version barrier (Sato and Chiba, 2005) of incompatibility between differently versioned classes and objects, the updating model must maintain a versioned view of the involved objects and classes. Gosh! utilizes a novel concept of Dynamic Correspondence Proxification, a combination of the two mechanisms In-Place Proxification and Correspondence Mapping which transform live objects and classes of former versions into proxies that delegate to the most recent versions. In-Place Proxification enforces shared identity and state across the version barrier, while Correspondence Mapping handles type conversion for crossing the version barrier. Details on Dynamic Correspondence Mapping can be found in prior work (Gregersen and Jørgensen, 2009). In this paper, we solely outline the architecture of Gosh!. Details on the architecture are provided in (Gregersen et al., 2012).

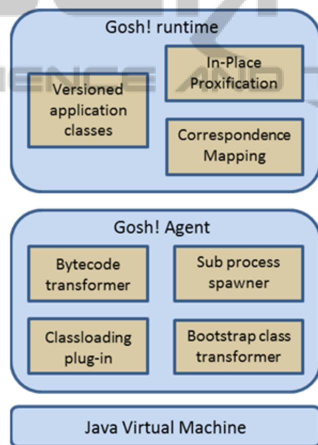

Figure 1: Architectural overview of Gosh!.

The architecture of Gosh!, shown in figure 1, features the following components:

- The *bootstrap-class-transformer* and *sub-process-spawner* components are responsible for statically transforming the JVM bootstrap-classes and to automatically spawn a new JVM process with the set of modified bootstrap-classes. This setup is necessary to make Gosh! transparent to the end-user, as the class instrumentation mechanism introduced in JDK 5.0 does not support instrumentation of bootstrap-classes on class loading.
- The *class-loading plug-in* component is used to integrate Gosh! with the class loading and resource management of different application frameworks. At present, Gosh! only provides an integration component for the NetBeans Platform, besides standard Java SE support. In general, the responsibility of these components is to deal with all the issues that cannot be handled simply by updating Java class files, i.e. reflecting changes made to application resources and configuration files.
- The *bytecode-transformer* component is responsible of instrumenting classes as they are loaded into the JVM. We distinguish between system classes that are made dynamic update-aware and application classes that are made dynamic update-enabled. Update-aware classes impose less run-time overhead than update-enabled classes. We make this distinction, because we consider it less likely that system classes are dynamically updated, as this would most likely include a dynamic update of the JVM. However, although system classes are not considered subject to dynamic updating they must be instrumented to accommodate changes to their possible subclasses.
- The *run-time* component implements the underlying dynamic updating model, which uses the In-Place-Proxification technique in combination with Correspondence Mapping to delegate requests to the most recent versions of updated classes. This component also ensures correct identity and equality preservation, handling of hashCode, thread synchronization, array-access handling for differently versioned objects etc. The core execution component is also responsible of transferring state from former versions to the new version. State is transferred using a thread-safe, non-blocking, lazy-state copying mechanism, which only transfers state when it is requested from within the new version. This ensures that the application stays responsive during dynamic updating as all state does not have to be transferred at once. In case all state had to be transferred at once, the end-user would experience a transition bump, where the application turns temporally inaccessible.

4 BENCHMARKING GOSH!

We have used SPECjvm2008 to measure the steady-state performance overhead introduced by Gosh! and JRebel 4.5.2. We chose to compare Gosh! with JRebel and not DCEVM, as Gosh! and JRebel are both application-level approaches whereas DCEVM is based on a modified Java HotSpot^{TM} VM. As shown by figure 2, Gosh! and JRebel are comparable in performance, both approaches also show similar

bottlenecks. The tests were performed so both Gosh! and JRebel identified the benchmark classes as update-enabled.

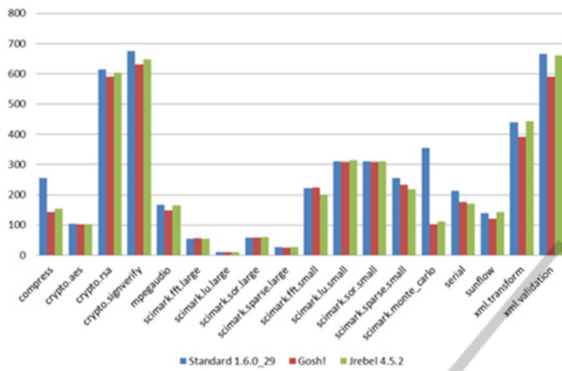

Figure 2: Gosh! vs. JRebel 4.5.2 [operations/min.].

Since the SPECjvm2008 test only allowed us to measure the steady-state performance overhead before updating we also designed a number of micro benchmarks to measure the run-time overhead imposed by newly inserted code after updating. The result of our recursive Fibonacci number benchmarks is shown in figure 3. The dynamic update simply renames the recursive method, thus simulating the insertion of a new method. The benchmark results show that Gosh! is faster than JRebel both before and after an update. Furthermore, the results also show that the runtime overhead remains constant for Gosh! after updating whereas it increases drastically for JRebel. Hence, Gosh! demonstrates that it scales for supporting continues updating.

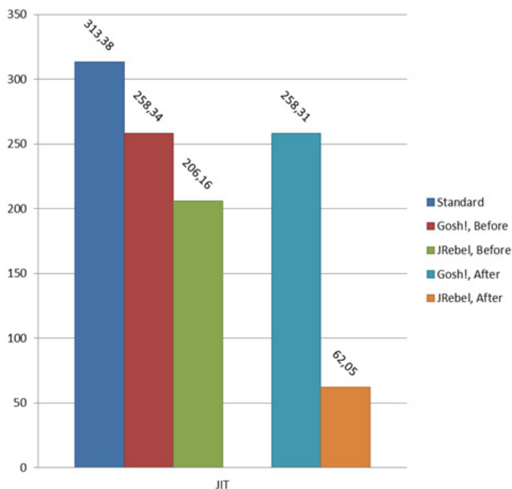

Figure 3: Recursive Fibonacci Benchmark [operations/min.].

5 EXPERIENCE

To evaluate practical application of Gosh!, we made four updates using five revisions of an in-house-developed version of the classical arcade game Breakout. The first version of the game contains 14 classes and 1.012 lines of code, which developed into 36 classes and 2.405 lines of code in the final version. The five revisions of the game contain many non-trivial code changes. A total of 120 code changes were found by manual inspection. Each code change has been classified according to the classification developed in (Gustavson, 2003). Table 2 summarizes the code changes found for successive revisions. The ID numbering of the code changes is not consecutive, as we have only listed the code changes that took place between successive revisions. A blank field in the table indicates that no occurrences of the code change were found. The last column in the table summarizes the frequency of each code change for all revisions. We included this column to show how often a particular code change occurs during development of the game.

Tabel 2: Code change analysis of Breakout.

ID	Code change description	R1-R2	R2-R3	R3-R4	R4-R5	%
6	Class added	2	9	3	3	14
30	Constructor implementation changed in class		1	1		2
33	Instance method added to class	3	11	2	8	20
34	Instance method removed from class	2				2
35	Instance method renamed in class	4				3
37	Instance method return type changed in class	4				3
38	Instance method implementation changed in class	12	11	3	10	30
44	Static method implementation changed in class	4				3
68	Instance field added to class		2		2	3
84	Interface added			1		1
120	Resource added [a]	8	1	4	8	18
121	Resource removed [a]				1	1

a. ID 120 and 121 is a refinement of ID 117 'Environment state change' in 0

The experiment showed that the Breakout game could be successfully updated from one revision to the next, however, under some circumstances the applied updating sequence resulted in a run-time behavior that was quite different from that of a cold restart of the game. Updates that resulted in different

behavior did so, because they introduced code changes that caused run-time phenomena. A classification of run-time phenomena in dynamic software updating was first introduced in (Gregersen and Jørgensen, 2011). The code changes listed in table 3 were herein identified as the cause of these phenomena. It is important to note that these code changes may cause run-time phenomena, but that it is not always the case. Whether run-time phenomena do occur is very dependent on the application's design and the time of updating.

Tabel 3: Run-time Phenomena.

ID	Code change description	Possible run-time phenomenon
7	Class removed	Phantom objects
8	Class renamed	Phantom objects / Lost State
16	Modifier abstract added to class	Phantom objects
6	Class added	Absent state
22	Super class of class changed	Absent state
68/71	Instance/static field added to class	Absent state
21	Modifier static removed from inner class	Absent state
70/73	Instance/static field type changed in class	Lost state
65	Static initialization impl. changed in class	Oblivious update
30	Constructor impl. changed in class	Oblivious update
114	Static field value changed	Broken assumption
38/44	Instance/static method impl. changed (e.g., conditional statement, method split / merged)	Broken assumption / Transient inconsistency

- *Phantom Objects* are live objects whose classes have been removed by a dynamic update. Whilst phantom objects will continue to exist in the system, their existence in the updated application will be void. Hence, if such objects are part of the existing application state, the updated application may try to reference them indirectly through, for instance, an array or a collection. Although removing classes is typically discouraged, there are situations where classes are either in-lined or renamed due to refactoring. For the DSU systems discussed in this paper, in-lining and class renaming corresponds to class-removed and class-added operations. Likewise, the use of dayfly classes (Lanza et al., 2005) is another example of class removals. Dayfly classes are classes that are typically created for evaluating a new idea and then removed shortly thereafter.
- *Absent State* refers to the situation where objects created in a former version lack state defined by the updated versions of their classes. Such state would typically have been created during a cold restart by an extra argument in a modified constructor.
- *Lost State* happens when an updated class makes binary incompatible changes to the type of a member field. E.g., change the field 'name' of type String to type Name. Given that it is not possible for the automatic state-transfer mechanism of Gosh! to automatically deduct how a changed type relates to a previously declared type, the run-time effect of changing the field type is that the field value for all existing objects of that class is lost and the new value is set to the default value.
- *Oblivious Update* refers to the situation where some or all features introduced in the new revision are missing after updating. That is, the run-time behavior of the updated application is different from that of a cool restart. Changing constructors to initialize new state fields is often the cause for oblivious updates, as constructor changes will not have any effect on already created objects.
- *Broken Assumption* may surface when constraints governing the interrelationship between program state and program logic change between successive revisions. If, for instance, the value of a member field, e.g. a counter, depends on some other member field, e.g. a constant, then changing either the value of the constant or the logic of the code maintaining this interdependency may break objects when moved to the new class. Exception-based program termination is often the result.
- *Transient Inconsistency* refers to the situation where an updated application is temporally brought into a run-time state that the new version of the application would never enter after a cold restart. If the updated application does not enter a valid run-time state in the new version after a finite period of time, it is said to be captured in an erroneous state. Erroneous state can be caused by a Broken Assumption that does not produce any run-time exceptions.

An interesting example of the Phantom Object and Lost State phenomena can be observed if we perform a dynamic roll-back by dynamically updating revision 4 back to revision 3 in the middle of a level. The resulting run-time effect of dynamic update is shown in figure 4. Here we see that the

special bricks introduced in revision 4 disappear after the roll-back to revision 3. This happens because revision 4 uses subclasses of the abstract parent class (Brick.class) to model special feature bricks, such as concrete bricks and bonus bricks that drop bonuses when hit. This roll-back corresponds to a class-removed code change, as the subclasses do not exist in revision 3. Hence, the roll-back resulted in a run-time behavior that is different from that of a cold restart, where the brick wall would have appeared solid consisting of only blue bricks. However, this effect is a Transient Inconsistency as the brick wall is drawn correctly when continuing to the next game level.

We observed during our experiments with dynamically updating of the Breakout game that the result of a dynamic update is highly dependent on application design. The roll-back goes through despite the occurrence of run-time phenomena, because of a loosely coupled design that uses a lookup service for storing the brick wall. An alternative design storing the brick wall in an array of type Brick[][] would result in program termination due to a null pointer exception, because the state-migration mechanism in Gosh! cannot map objects of subclasses for special feature bricks in revision 4 to any objects in revision 3, as the subclasses do not exist here. Hence, the state-migration mechanism will instead insert null references in place of the original special brick objects in the array. It is these null-references that cause program termination due to a null pointer exception when traversing the array. More examples on the run-time phenomena and their causes are given in (Gregersen and Jørgensen, 2011).

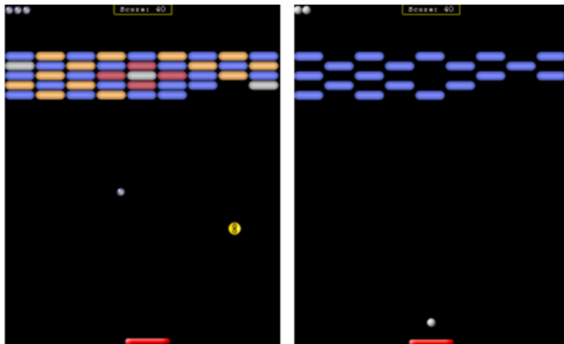

Figure 4: Disappearing objects after class removals.

6 CONCLUSIONS

In this paper, we have provided an overview of the current state-of-the-art of the DSU systems targeting Java, by comparing the set of code changes they support. The comparison shows that Gosh! is currently the most comprehensive DSU system available. Furthermore, we have benchmarked Gosh! against the only commercially available DSU system JRebel and shown that Gosh! delivers comparable run-time performance before updating and considerable better performance after updating. Whereas JRebel's updating model introduces a significant overhead for handling changed code, Gosh!'s updating model scales and continues to perform with the same constant run-time overhead. Hence, Gosh! shows the capability to provide support for dynamic updating of long-lived applications, like application- and web-servers. To evaluate Gosh!, we made four updates using five revisions of an in-house developed version of the classical arcade game Breakout. The experiment showed that it was possible to incrementally update the consecutive revisions of the Breakout game. However, what the experiment also showed was that dynamic updating may result in so-called run-time phenomena. I.e., situations where the run-time behavior of the updated application diverges from the behavior expected after a cold restart. Hence, to increase predictability of DSU systems there is a need for creating dynamic impact analysis tools that can determine whether code changes differentiating successive revisions may potentially lead to manifestation of run-time phenomena or not. Dynamic analysis is necessary as both the run-time state and the time of updating have significant impact on the result of an update, hence static impact analysis alone cannot determine whether a dynamic update will be successful, it can only identify potential risks of run-time phenomena. The advent of dynamic analysis tools will, among other things, determine the future success and feasibility of dynamic updating for mission critical software systems.

REFERENCES

Altekar, G., Bagrak, I., Burstein, P., Schultz, A., 2005. OPUS: Online patches and updates for security. In *Proc. USENIX Security*.

Arnold, J., Kaashoek, F., 2009. Ksplice: Automatic rebootless kernel updates. *In Proc. EuroSys*.

Baumann, A., Appavoo, J., Silva, D. D., Kerr, J., Krieger, O., Wisniewski, R. W., 2005. Providing dynamic update in an operating system. *In Proc. USENIX Annual Technical Conference*.

Baumann, A., Appavoo, J., Wisniewski, R. W., Silva, D. D., Krieger, O., Heiser, G., 2007. Reboots are for

hardware: challenges and solutions to updating an operating system on the fly. *In Proc. USENIX Annual Technical Conference*.

Bierman, G., Parkinson, M., Noble, J., 2008. UpgradeJ: Incremental typechecking for class upgrades. *In Proc. ECOOP*.

Chen, H., Chen, R., Zhang, F., Zang, B., Yew, P. C., 2006. Live updating operating systems using virtualization. *In Proc. VEE*.

Chen, H., Yu, J., Chen, R., Zang, B., Yew, P., 2007. POLUS: A POwerful Live Updating System. *In Proc. ICSE*.

Dmitriev, M., 2001. *Safe Evolution of Large and Long-Lived Java Applications*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

Gustavson, J., 2003. A Classification of Unanticipated Runtime Software Changes in Java. *In Proc. ICSM*.

Gregersen, A. R., Jørgensen, B. N., 2009. Dynamic update of Java applications—balancing change flexibility vs programming transparency. *In J. Softw. Maint. Evol.*: Res. Pract. 21.

Gregersen, A. R., Jørgensen, B. N., 2011. Run-time Phenomena in Dynamic Software Updating: Causes and Effects. *In Proc. IWPSE-EVOL*.

Gregersen, A. R., Hadaytullah, Koskimies, K., Jørgensen, B. N., 2012. An Integrated Platform for Dynamic Software Updating and its Application in Self-* systems. *In Proc. SCET*.

Kabanov, J., 2010. JRebel Tool Demo. *In Proc. Bytecode 2010*.

Lanza, M., Ducasse, S., Gall, H., and Pinzger, M., 2005. CodeCrawler: an information visualization tool for program comprehension. *In Proc. ICSE*.

Lehman, M. M., 1997. Laws of Software Evolution Revisited, pos. paper, EWSPT96, Oct. 1996, Lecture Notes in Computer Science, Vol. 1149, *Springer-Verlag*.

Makris, K., Bazzi, R., 2009. Multi-threaded dynamic software updates using stack reconstruction. *In Proc. USENIX Annual Technical Conference*.

Makris, K., Ryu, K. D., 2007. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. *In Proc. EuroSys*.

Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J. F., 2000. Runtime support for type-safe dynamic Java classes. *In Proc. ECOOP*.

Microsoft Corporation. Edit and continue. http://msdn2.microsoft.com/en-us/library/bcew296c.aspx, 2008.

Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M., 2006. Practical dynamic software updating for C. *In Proc. PLDI*.

Orso, A., Rao, A., Harrold, M. J., 2002. A technique for dynamic updating of Java software. In Proc. ICSM.

Ritzau, T., Andersson, J., 2000. Dynamic deployment of Java applications. *In Proc. Java for Embedded Systems Workshop*.

Sato, Y., Chiba, S.,2005. Loosely-separated "Sister" Namespaces in Java. *In Proc. ECOOP'05*.

Soules, C., Appavoo, J., Hui, K., Silva, D. D., Ganger, G., Krieger, O., Stumm, M., Wisniewski, R., Auslander, M., Ostrowski, M., Rosenburg, B., Xenidis, J., 2003. System support for online reconfiguration. *In Proc. USENIX Annual Technical Conference*.

Subramanian, S., Hicks, M., McKinley K. S., 2009. Dynamic software updates: a VM-centric approach. SIGPLAN No. 44, 6.

Sun Microsystems. Java Platform Debugger Architecture, 2004. This supports class replacement. See http://java.sun.com/javase/6/docs/technotes/guides /jpda/.

Würthinger, T., Wimmer,C. and Stadler, L. 2010. Dynamic code evolution for Java. *In Proc. PPPJ*.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS