Dynamic Software Updates for Accelerating Scientific Discovery

Dong Kwan Kim, Myoungkyu Song, Eli Tilevich, and Shawn A. Bohner

Center for High-End Computing Systems (CHECS)
Dept. of Computer Science, Virginia Tech
Blacksburg, VA 24061, USA
{ikek70,mksong,tilevich,sbohner}@cs.vt.edu

Abstract. Distributed parallel applications often run for hours or even days before arriving to a result. In the case of such long-running programs, the initial requirements could change after the program has started executing. To shorten the time it takes to arrive to a result when running a distributed computationally-intensive application, this paper proposes leveraging the power and flexibility of dynamic software updates. In particular, to enable flexible dynamic software updates, we introduce a novel binary rewriting approach that is more efficient than the existing techniques. While ensuring greater flexibility in enhancing a running program for new requirements, our binary rewriting technique incurs only negligible performance overhead. We validate our approach via a case study of dynamically changing a parallel scientific simulation.

Key words: Dynamic Software Updates, Time-to-Discovery, Computationally-Intensive Applications, JVM HotSwap, Bytecode Enhancement

1 Introduction

Scientific computing is an interdisciplinary research area that uses computer technologies to analyze mathematical models for computationally demanding problems, including forecasting the weather, predicting earthquakes, and simulating molecular dynamics. Despite the ever increasing computing power, scientific computing applications are often long-running, taking hours or even days to arrive to a result, due to the tremendous amounts of involved computations. An effective approach to reducing the computing time in scientific programs is parallel processing, particularly using compute clusters and computational grids.

In a long-running application, the initial scientific requirements could change while the execution is in progress. To realize the changed requirements, a standard approach requires stopping the running application, changing the code, and restarting the application. However, this maintenance approach does not utilize the computing resources most effectively, as it leads to repeating some of the computation.

This work is concerned with perfective maintenance required to address changes in requirements rather than corrective maintenance required to address

defects. We assume that the running program is correct, but needs to change to meet some newly-discovered requirements. We are not considering the problem of detecting and correcting program defects, which is addressed elsewhere in the research literature [1].

This work targets distributed computationally-intensive applications that use the Java™technology as a means to operate in heterogeneous environments. Successful applications of the Java technology to the domain of distributed parallel computation include heterogeneous, Java-based, computational grids [2]. The Java Virtual Machine (JVM) provides an advanced virtual execution environment on multiple platforms. The adaptive optimization capabilities of Just-In-Time (JIT) compilers make the JVM suitable for executing programs written in scientific computing languages, including X10 [3], and possibly other emerging languages such as Fortress [4].

Although the JVM features the HotSwap API [5], which replaces loaded classes in a running application, the signature of a replaced class must remain the same, allowing only method body changes. This, in turn, constrains the programmer modifying the swapped classes. This paper shows how these HotSwap constraints can be overcome to allow the programmer to update classes without restrictions. To that end, this paper presents a novel bytecode rewriting and code generation approach, enabling the use of the standard HotSwap to replace the changed code in a running JVM. The approach leverages *Binary Refactoring*, a technique we introduced [6] that applies semantics-preserving transformations to the binary representation of a program. The flexible and efficient dynamic updates enable the programmer to perfect a running application at will. The resulting incremental perfective maintenance model can reduce time-to-discovery when fine-tuning a distributed computationally-intensive application.

This paper presents a solution to the problem of updating computationally intensive applications dynamically and contributes a novel dynamic update method that can perfect long-running, distributed, JVM-based applications for new requirements, thereby shortening their time-to-discovery; and a new binary rewriting technique that enables the enhancement of performance-sensitive applications, with minuscule performance overhead.

The rest of this paper is structured as follows. Section 2 details our approach to updating computationally-intensive software dynamically. Section 3 evaluates the flexibility and efficiency of our approach. Section 4 compares our approach with the existing state of the art. Section 5 discusses future work directions and presents concluding remarks.

2 Updating Computationally Intensive Applications Dynamically

Next we describe our flexible and efficient dynamic software updating system for computationally intensive applications.

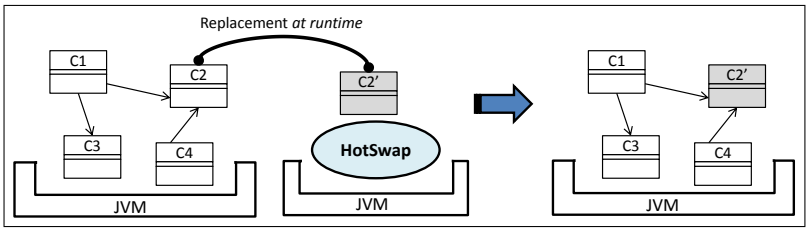

Fig. 1. JVM HotSwap facility.

2.1 Enhancing JVM HotSwap Using Bytecode Rewriting

Fig. 1 shows how the JVM HotSwap reloads class *C2'* on the fly. The replaced application with four classes and the HotSwap program with a newer version *C2'* execute on two different JVMs; the JVM running the target application needs to start with the appropriate debugging options and the HotSwap module connects the JVM with its hostname and port number. The rightmost part of Fig. 1 shows the application has the new version *C2'*.

Although the HotSwap API can replace loaded classes in a running application, the signature of a replaced class must remain the same, and only method bodies could change. Thus, adding new methods, fields, or constructors, or even changing the signatures of existing methods or fields will render a class invalid for HotSwap, thus hindering the programmer from updating programs at runtime. To remove these constraints, our dynamic updating approach leverages the ability of the JVM to load classes at runtime and uses bytecode rewriting and code generation.

Fig. 2 shows our binary rewriting which introduces an indirection to a target class using the Proxy Pattern. This rewrite leverages advanced optimization capabilities of modern JVMs to inline the indirected functionality, making the rewrite applicable for performance-sensitive applications [7]. The original class *A* is translated into the proxy *A* and its superclass *Super_A*. While the class name and the method signatures of the original and proxy classes remain the same, the method bodies are different; the overloading methods of the proxy class invoke the overloaded methods of the superclass. The code snippet in the

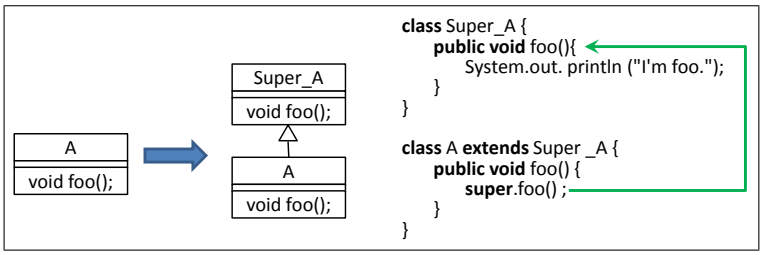

Fig. 2. Our binary rewriting to introduce indirections.

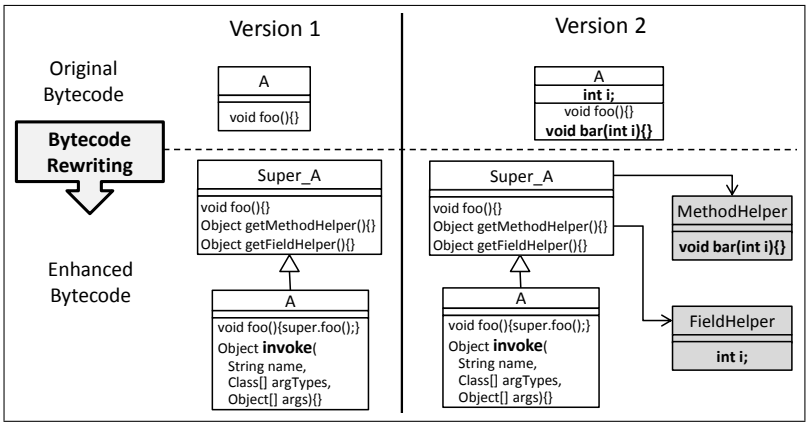

Fig. 3. Adding new members to the class *A* using the special helper classes.

right shows that the proxy class *A* inherits methods from the superclass *Super_A* and the call to the method *foo()* is delegated to the superclass.

Our approach first rewrites an original program for updatability and then changes it for new requirements dynamically. In order to make a program updatable, our bytecode enhancer transforms the bytecode using the techniques described above. This approach supports a wide-range of changes to the reloaded classes, without violating the constraints imposed by the JVM HotSwap API.[1] Fig. 3 describes an example of our approach. Suppose that class *A* needs to be updated with another version. Since the second version of *A* is structurally different from the first version, the current JVM HotSwap implementation cannot reload the second version of *A*. Our approach makes new classes for added fields and methods to provide the flexibility. We called them helper classes. There are two helper classes in Fig. 3; *MethodHelper* is for the new method *bar(int i)* and *FieldHelper* is for the new field *int i*. To make intended changes to the running application, we can reload the proxy *A* and its superclass *Super_A* using HotSwap. It is obvious that JVM loads two helper classes when it reloads *Super_A*.

2.2 Updating Scientific Applications on the Fly

Once a program is made updatable at the bytecode level, the JVM HotSwap can replace at runtime the program's classes with their newer versions containing structural differences. Furher, the HotSwap facilities are used in exactly the same way as shown in Fig. 1. Fig. 4 illustrates the modules and control flow of our dynamic updating system. This system consists of the class differencing and bytecode rewriting modules.

To generate the helper classes, our approach identifies the structural changes between the two versions of a class. The class differencing algorithm shown in

[1] Our approach does not support the dynamic updates that change the inheritance hierarchy–these changes are too substantial to be supported by rewriting bytecode.

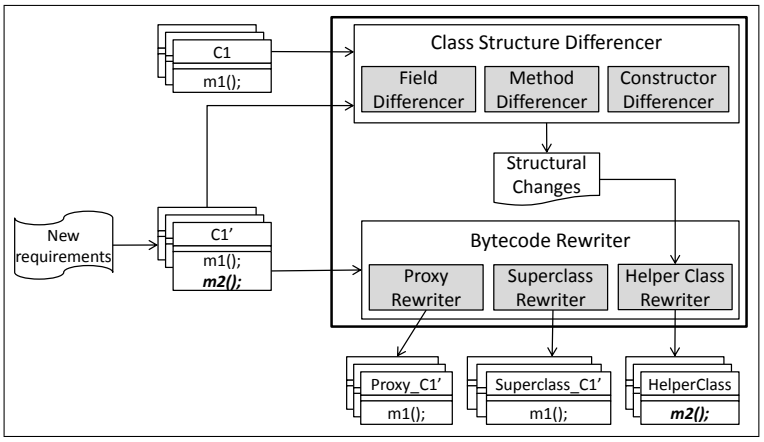

Fig. 4. Our dynamic software update system-subsystems and control flow.

Fig. 5 takes two versions of the same program as input and returns a collection of differences in fields, constructors, and methods. This algorithm simply compares the fields, methods, and constructors of the classes by using the Java Reflection API. To find out field differences, the differencing algorithm compares the modifier, type, and name of a field. The method differences are identified by examining the modifier, return type, name, and parameter types of a method and constructors are distinguished by their modifier and parameter types.

Next we provide a more formal treatment of our binary rewriting techniques using superclasses and proxies. The double vertical bar ($\|$) specifies pre- and post-conditions. In $\frac{X}{Y}$, X denotes the class hierarchy of the original class before the enhancement, while Y denotes its new hierarchy after the enhancement has been performed. In Fig. 6, c is an original class, transformed into a proxy and with the added superclass. The new superclass c_{vs} is inserted between the proxy c_{proxy} and the initial superclass s of the original class c. Fig. 7 depicts how the indirection works for methods and constructors. Fig. 8 details how our approach introduces an indirection when accessing non-private fields. $private <_v V$ denotes the visibility V which is stronger than *private* visibility.

3 Case Study: Updating a Molecular Dynamics Simulation System Dynamically

To demonstrate the efficiency of our approach, we compared the total execution time of a Successive Over-Relaxation (SOR) [8] program with that of its rewritten version. The measurements were conducted on a compute cluster, with each node running a dual processor AMD Opteron 240 (1.4Ghz), 1GB RAM, CentOS version 4.2, JDK version 1.5.0, connected by Myrinet (4Gbit). Fig. 9 shows the total overhead of the rewritten version never exceeds 2%.

To assess the applicability of our approach to more realistic programs, we used a parallel Molecular Dynamics Simulation (MDS) program [9, 10] which

```
INPUT: A set C = {(c_v1^1, c_v2^1), (c_v1^2, c_v2^2), ..., (c_v1^p, c_v2^p)} of class pairs to be compared.
OUTPUT: A collection of differences of fields, constructors, and methods
1  while (a set C is not empty) do
2    //Compare fields of classes
3    fieldsOfOldClass ⟵ c_v1^i.allFields(), fieldsOfNewClass ⟵ c_v2^i.allFields()
4    for (fieldsOfNewClass is not empty) do
5      eachFieldOfNewClass ⟵ fieldsOfNewClass.nextItem()
6      for (fieldsOfOldClass is not empty) do
7        eachFieldOfOldClass ⟵ fieldsOfOldClass.nextItem()
8        if ( sig . of eachFieldOfOldClass == sig. of eachFieldOfNewClass) then
9          isSameField = true; break;
10       end if
11     end for
12     if (NOT isSameField) then
13         differentMembers.addElement(eachFieldOfNewClass)
14     end if
15   end for
16   //Compare constructors and methods of classes
17   methodsOfOldClass ⟵ c_v1^i.allMethods(), methodsOfNewClass ⟵ c_v2^i.allMethods()
18   for (methodsOfNewClass is not empty) do
19     eachMethodOfNewClass ⟵ methodsOfNewClass.nextItem()
20     for (methodsOfOldClass is not empty) do
21       eachMethodOfOldClass ⟵ methodsOfOldClass.nextItem()
22       if ( sig . of eachMethodOfOldClass == sig. of eachMethodOfNewClass) then
23         isSameMethod = true; break;
24       end if
25     end for
26     if (NOT isSameMethod) then
27         differentMembers.addElement(eachMethodOfNewClass)
28     end if
29   end for
30 end while
```

Fig. 5. The *ClassDifferencing* algorithm.

was deployed on Ibis [2], a Java-based grid programming environment. Among other services, Ibis provides a Java API for MPI-like message passing among cluster nodes.

A set of interfaces, $I = \{i_1, i_2, \ldots, i_i\}$
$VSuper(c \| c_{proxy}, c_{vs})$: Class c is transformed into c_{proxy} and c_{vs}.
c: refactored class, c_{proxy}: proxy class of c, c_{vs}: new superclass of c

$$VSuper(c \| c_{proxy}, c_{vs}) = \frac{c \ extends \ s \ implements \ I}{c_{proxy} \ extends \ c_{vs}, \ c_{vs} \ extends \ s \ implements \ I}$$

Fig. 6. Indirection using superclasses.

P_{vs} ‖...‖ denotes the rewriting by our approach.
//*The transformation of constructors*
P_{vs} ‖ *public* $k(T_1, \ldots, T_n)$ *throws* $C_1, \ldots, C_i$‖ =
    public $k(T_1, \ldots, T_n)$ throws $C_1, \ldots, C_i$ { super($T_1, \ldots, T_n$); }
//*The transformation of methods*
P_{vs} ‖ *public* T $m(T_1, \ldots, T_n)$ *throws* $C_1, \ldots, C_i$‖ =
    public T m($T_1, \ldots, T_n$) throws $C_1, \ldots, C_i${
        if (the return type of m is *void*) super.$m(T_1, \ldots, T_n)$;
        else return super.$m(T_1, \ldots, T_n)$;
    }

Fig. 7. Indirecting constructors and methods.

//*Access the superclass's non − private fields*
G_{vs} ‖...‖ represents the generation of getters and setters for fields.
G_{vs} ‖ *private* $<_v V\ T\ x$‖ =
        private $<_v$ V T getX() { return x; }
        private $<_v$ V void setX(T x) { this.$x = x$; }
//*Access non − private fields via a proxy*
P_{vs} ‖ *private* $<_v V\ T\ x$‖ =
        private $<_v$ V T getX() { return super.getX(); }
        private $<_v$ V void setX(T x) { super.setX(x); }

Fig. 8. Indirecting the superclass's non-private fields.

We updated the MDS program dynamically twice, updating the thermostat algorithm and the number of molecules. The thermostat algorithm maintains or rescales the temperature constant of a molecular system by increasing or decreasing the velocity of the molecules. Therefore, the selection of an appropriate

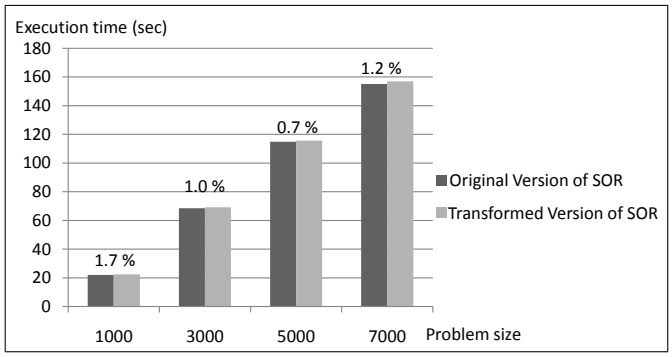

Fig. 9. Refactoring overhead on Successive Over-Relaxation.
x-axis: the problem size;
y-axis: the total execution time of both the original and the enhanced versions.

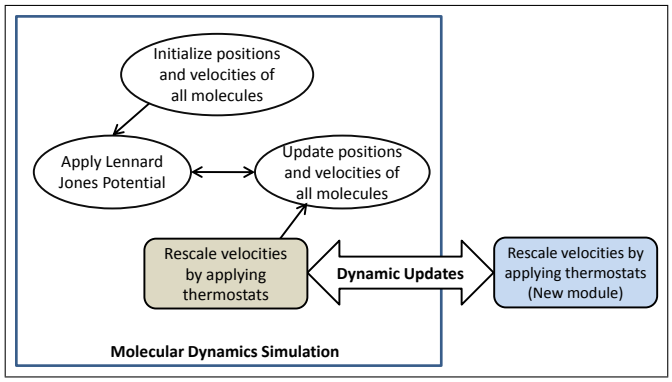

Fig. 10. Updating the rescaling module of a molecular dynamics simulation.

thermostat method depends on the molecular system in use. Also the initial number of the molecules may need to change during the simulation. Fig. 10 depicts the main modules of an MDS program and the thermostat module that are updated dynamically. Table 1 summarizes the aforementioned scenarios, which motivate dynamic changes.

While the changes above may seem simple, without dynamic updating facilities, they would require stopping the parallel execution and losing valuable computing resources. Furthermore, these updates could not be accomplished by using HotSwap alone. In fact, trying to use HotSwap for these updates would throw an exception terminating the program's execution. Finally, these changes are a natural consequence of delivering solutions under tight deadlines. It is not always possible to put enough care into designing a distributed parallel application, so that it always satisfies the requirements of different users.

Table 1. Changes to the Molecular Dynamics Simulation.

Updates	Requirements	Implementations
Thermostat algorithm	Rescale velocities of molecules by replacing the thermostat algorithm	Adding a new method `rescale(mol[] m, int size)` and fields
The number of molecules	Increase/decrease the number of molecules to be simulated	Adding a new method `updateNumOfMols(int size)`

4 Related Work

A significant amount research has been conducted in dynamic software updating via program transformation [11, 12], custom virtual machines or runtime libraries [13, 14], and new language constructs [15, 16] such as Aspect-Oriented Programming (AOP) [17].

Orso *et al.*'s technique [11] and Bialek *et al.*'s system [12] transform the code to enable its dynamic updates. Unlike our approach facilitates JVM HotSwap, both approaches do not use HotSwap and consider an efficient implementation of the proxy pattern for performance-sensitive applications. Furthermore, although custom virtual machines might be more powerful in dynamic software updates, they could lead to a severe interoperability issue in a heterogeneous computing environment. Like Warth *et al.*'s Expanders [15] and Bierman *et al.*'s UpgradeJ [16], dynamic updates can be provided as new language features or a service of middleware systems. While they can express explicitly changes at the code level, the programmer is required to learn new language constructs or tools. Similar to program transformation, AOP-based approaches need to insert dynamic update modules, usually aspects, into a target application before the application is executed [18–20]. Table 2 compares the proposed approach with closely related work. While these approaches to dynamic updates are powerful and effective, none of them is applied and tested for computationally intensive applications such as scientific and bioinformatics programs.

Table 2. Comparison to related work on dynamic updating for Java software(supported:+, unsupported:-, and partially supported:+/-).

Criteria	Orso[11]	Bialek[12]	Mal.[14]	Lee[21]	Pre.[18]	Ours
Use of Standard						
-Standard virtual machine	+	+	-	+	-	+
-HotSwap	-	-	-	-	-	+
-No coding constraints	+	+	+	-	+	+
-No runtime library required	+	+	-	-	-	+
Flexibility						
-Adding fields/methods	-	+	+	+	+	+
-Update of fields/methods	-	+	+	+	+	+
-No source modification	+	+	+	+	+	+
Efficient code	-	-	+	+/-	-	+

5 Future Work and Conclusions

The flexibility and efficiency of our approach open a slew of future work directions, including the application of our approach to large scale grid applications, self-adapting systems, and autonomic computing.

We have presented a new binary rewriting approach for supporting flexible and efficient dynamic updates of JVM-based, distributed, computationally-intensive applications. Our approach to dynamic updating works with standard JVMs and their built-in HotSwap facility to reload classes at runtime. The performance and flexibility advantages of our approach make it promising for reducing the time-to-discovery in long-running scientific applications.

References

1. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: treating bugs as allergies—a safe method to survive software failures. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating Systems Principles, ACM (2005) 235–248
2. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient Java based grid programming environment. Concurrency and Computation: Practice and Experience **17**(7-8) (June 2005) 1079–1107
3. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. **40**(10) (2005) 519–538
4. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Jr., G.L.S., Tobin-Hochstadt, S.: The Fortress language specification. Sun Microsystems, Inc. (March 2007)
5. Sun Microsystems, Inc.: Java HotSwap, `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html`
6. Tilevich, E., Smaragdakis, Y.: Binary refactoring: Improving code behind the scenes. In: Proceedings of International Conference on Software Engineering (ICSE). (May 2005) 264–273
7. Kim, D.K., Tilevich, E.: Overcoming JVM HotSwap constraints via binary rewriting. In: First ACM Workshop on Hot Topics in Software Upgrades, ACM (2008)
8. Ortega, J.M.: Introduction to Parallel Vector Solution of Linear Systems. Plenum Press, New York, NY, USA (1988)
9. Rapaport, D.C.: The Art of Molecular Dynamics Simulation. Cambridge University Press, New York, NY, USA (1996)
10. Kumar, A.: Molecular Dynamics Simulations `http://www.personal.psu.edu/auk183/MolDynamics/Molecular%20Dynamics%20Simulations.html`.
11. Orso, A., Rao, A., Harrold, M.J.: A technique for dynamic updating of Java software. Proceedings of the International Conference on Software Maintenance (ICSM'02) (October 2002)
12. Bialek, R.P.: Dynamic updates of existing Java applications. In: Ph.D. Thesis, the University of Copenhagen. (2006) 1–216
13. Gharaibeh, B., Dig, D., Nguyen, T.N., Chang, J.M.: dReAM: Dynamic refactoring-aware automated migration of Java online applications. Technical Report, Iowa State University (August 2007)
14. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime support for type-safe dynamic Java classes. Proceedings of the 14th European Conference on Object-Oriented Programming **1850** (June 2000) 337–361
15. Warth, A., Stanojević, M., Millstein, T.: Statically scoped object adaptation with Expanders. In: OOPSLA '06. (2006) 37–56
16. Bierman, G., Parkinson, M., Nob, J.: UpgradeJ: Incremental typechecking for class upgrades. In: ECOOP. (July 2008)
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwing, J.: Aspect-oriented programming. In: ECOOP, Springer-Verlag (1997)
18. Previtali, S.C., Gross, T.R.: Dynamic updating of software systems based on aspects. Proceedings of the 22nd IEEE International Conference on Software Maintenance (September 2006) 83 – 92
19. Gustavsson, J., Staijen, T., Assmann, U.: Runtime evolution as an aspect. First International Workshop on Foundations of Unanticipated Software Evolution (2004)
20. Yang, Z., Cheng, B.H.C., Stirewalt, R.E.K., Sowell, J., Sadjadi, S.M., McKinley, P.K.: An aspect-oriented approach to dynamic adaptation. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, ACM (2002) 85–92
21. Lee, Y.F., Chang, R.C.: Java-based component framework for dynamic reconfiguration. IEE Proceedings - Software **152**(3) (June 2005) 110–118