

# Extracting Updating Aspects from Version Differences

Susanne Cech Previtali  
Institute of Computer Systems  
ETH Zurich  
Switzerland

Thomas R. Gross  
Institute of Computer Systems  
ETH Zurich  
Switzerland

## ABSTRACT

Dynamic software evolution represents a viable technique to update software systems at run-time. On-the-fly updating is particularly helpful for systems that must be continuously available and up-to-date. Updates consist of several incremental changes that must be applied to a system. These individual changes exhibit symptoms similar to crosscutting concerns: they are typically scattered across several classes. We therefore leverage aspect technology and use aspects to encapsulate the changes constituting an update. In addition, dynamic aspect systems can serve as a practicable platform to inject updates at run-time. In this paper, we present how we extract the updating aspects for a dynamic aspect system. We detail the algorithms necessary to generate such updating aspects that rely on the computation of the dependencies between the changes to determine the granularity of an aspect.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Version control*; D.3.4 [Software]: Compilers; F.3.2 [Lodges and Meanings of Programs]: Semantics of Programming Languages—*Program analysis*

## General Terms

Design, Languages

## Keywords

Aspect extraction, software evolution

## 1. INTRODUCTION

The deployment of software updates traditionally requires terminating running systems to replace the current software

---

The work presented in this paper was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LATE Linking Aspect Technology and Evolution Workshop, April 1st, Brussels, Belgium

Copyright 2008 ACM 978-1-60558-147-7/08/04 ...\$5.00.

version with the new one. Many software systems, however, must be continuously available and up-to-date. Rather than upgrading software systems to the next version based on the installation of a completely new binary version, software systems could be updated during their execution. Dynamic software evolution refers to bringing a running application to the latest version without stopping the application. As a consequence, the update must be expressed in terms of incremental changes (i.e., addition, removal, and modification of code).

We have described an approach for the dynamic evolution of object-oriented systems in an earlier paper [6]. Software updates should be modularizable and, when employed, preserve the type-safety of the running application. Aspect technology has proven viable for modularizing crosscutting concerns in object-oriented systems. Since updates affect various, seemingly unrelated parts of the system, we propose to handle updates in a similar manner as crosscutting concerns. Using aspects as the basic updating unit, we can deploy a dynamic aspect system (e.g., [1, 2, 3, 9, 10, 11, 12]) to achieve dynamic evolution of software. As we require the aspect system to atomically weave and deploy the updating aspects, the type-safety is not violated.

In earlier work, we discussed possible update scenarios to validate the approach of aspects-based updating [5]. In this paper, we elaborate on how to extract updating aspects from the differences of two complete versions of a Java application and detail issues with regard to run-time code adaptation. This paper is structured as follows: Section 2 gives an overview of the software evolution system. Section 3 presents how updates can be represented with aspects. Section 4 shows how updating aspects are generated. Section 5 discusses the proposed solution, and Section 6 concludes.

## 2. SYSTEM OVERVIEW

In this section, we give an overview of the software evolution system focusing on the parts relevant to aspect extraction. Our software evolution system accomplishes dynamic evolution [4] and thus changes an application during its execution. To compute the necessary updates, the system compares *statically* two complete versions of a Java application, deduces their differences, and expresses the necessary changes as aspects, which are then applied *dynamically*. As a consequence, the evolution system consists of two independent systems, a compile-time and a run-time system (see Figure 1). In the following, we explain the significant parts of both systems and refer to [6] for a more detailed description.

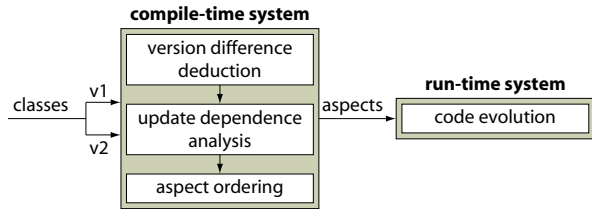


Figure 1: Software evolution system architecture.

### Compile-time system.

The goal of the compile-time system (or system for short when clear from the context) is to automate the three steps depicted in Figure 1:

(1) *Version difference deduction.* All classes (i.e., class files) constituting the old and the new version of the application are compared to deduce the differences. Classes and (recursively) fields and methods are compared to compute (a) the sets of unchanged, modified, added, and removed classes, fields, and methods as well as (b) the kind of modifications.

(2) *Update dependence analysis.* The system must determine the changed methods that must be encapsulated in an aspect and therefore, the system distinguishes between changes to the *implementation* and the *specification* of a class [8, 13]: The change of a method body is not visible from outside the class and consequently alters only its implementation. All changes to fields and methods that are accessible from other classes may affect the specification of that *supplier* class. For example, the addition or removal of methods, as well as modifications to argument and return types, alter the specification of a class. Specification changes always trigger changes in the *client* classes, i.e., the users of the supplier class. In other words, for a change in a supplier class to become effective, the *dependent* changes in client and supplier must be updated together and consequently are grouped in a single aspect.

(3) *Aspect ordering.* We allow the user to insert the updating aspects at any time during the execution of the program. To guarantee that outdated methods will no longer be executed once updated code is installed, the system imposes a weaving order on the updating aspects analyzing the method call graph. Because dependent changes are grouped in one aspect and are woven atomically, aspects are regarded as super-nodes in the call graph. By identifying the strongly-connected components, cyclic method call chains can be collapsed, resulting in a (directed) acyclic graph. Ordering is achieved by performing a topological sort of this simplified graph.

### Run-time system.

The updating aspects are woven into the running application to achieve dynamic software evolution. We selected PROSE [9, 10] as the aspect system as it supports truly dynamic weaving, i.e., the aspects are not provided on system start-up. A PROSE aspect is a Java class that contains one or several crosscuts; this allows us to define several updates in one aspect. A *crosscut* consists of an *advice* code and an (optional) pattern expression called a *pointcut*. Advice is the code to be added to the application. The pointcut expression selects the methods to be affected. PROSE in-

tegrates the advice code by replacing the original method body by the code in the advice. PROSE weaves each aspect atomically, such that all redefined methods declared in the aspect are replaced with the new code. PROSE allows aspects to be prioritized to establish an order in which an aspect will be woven relative to other aspects.

## 3. UPDATE REPRESENTATION

In this section, we describe how we represent updating aspects for the dynamic aspect system PROSE. To ease understanding, we will give examples both for PROSE and AspectJ [14]. In earlier work [5], we described different update scenarios between a client and a supplier class and their corresponding updating aspects with AspectJ. Note that updates that alter the inheritance structure of classes and interfaces or that change the memory layout of objects can only be handled by a static weaver. For example, the addition of a field requires existing objects to be adapted to the new representation and the new field to be initialized. In this paper, we focus on method and field evolution not requiring object adaptation and discuss issues with regard to dynamic code adaptation. In the following, we first present how changes to the implementation of a method may be handled. Then we consider specification changes that affect at least two methods.

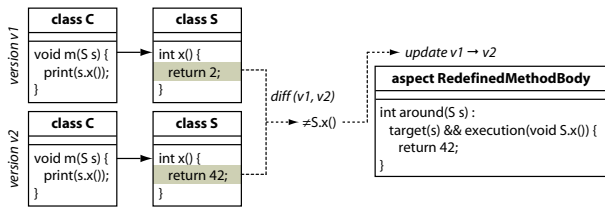
### Implementation changes.

Changes to the implementation of a method affect its body. An aspect can capture this change and replace the original method body with an around-advice. Figure 2 shows that the body of method `x()` of class `S` changes in the new version. The corresponding updating aspect replaces the original method with the body of the new version. The PROSE aspect (see Figure 2 below) declares a `MethodRedefineCut` containing two methods: The first method, `METHOD_ARGS()`, provides the body of the new version as well as one part of the pointcut expression: the arguments of `METHOD_ARGS()` may specify the method’s argument types where the first argument is reserved for the target type. The second method, `pointCutter()`, specifies the names of the class and the method. Note that as only the implementation changes, the aspect remains local to the advised class.

### Specification changes.

Modifications of the specification of a class include the addition of new methods and changes of return or argument types. These changes do not remain local to the (supplier) class, instead they are effected and continued in other (client) classes. Figure 3 shows that class `S` introduces a new argument to its method `x()` in the new version. Class `C` alters the method body `m()` to adapt to the change of the other method. The changes crosscut both classes `C` and `S`, and the crosscutting character of the aspect allows us to encapsulate both changes.

With a static weaving approach such as AspectJ, modifications concerning the specification of a class (e.g., added methods) or a method (e.g., changed argument or return types) are declared using an intertype member declaration in the aspect. In Figure 3, for instance, the aspect declares a new intertype method `x(int)` to be supported by `S` and redefines method `m()` of `C`. Dynamic aspect systems, on the other hand, do not support intertype member declaration



```

public class RedefinedMethodBody extends DefaultAspect {
    public Crosscut c = new MethodRedefineCut() {
        /* Match S.*(*) and redefine body */
        public int METHOD_ARGS(S target) {
            return 42;
        }
        /* Match S.x(*) */
        protected PointCutter pointCutter() {
            return (Within.subType(S.class)).AND(Within.method("x"));
        }
    };
}

```

**Figure 2: Change of the implementation: AspectJ aspect in upper right, PROSE aspect below.**

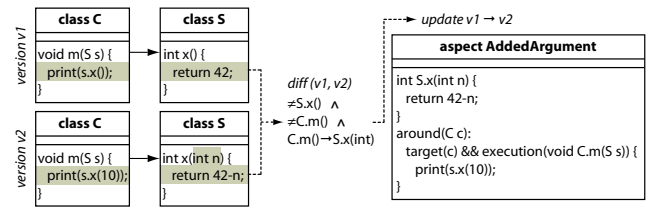
and as a consequence, the members must be declared as actual aspect members. Using PROSE, for instance, we declare the same method (`x(int)`) in the aspect and explicitly add the original class as a first argument to the method to be able to refer to the target object (see Figure 3). As the modifications of the specification are reflected in the calling method, the aspect includes an around-advice for that caller.

Note that specification changes in a supplier class may not only yield implementation changes in a client class, but also specification changes that lead to further adaptations in different client classes. In general, updates may encompass any number of methods. For example, a client changed a method body to call an added method that itself calls another added method. Furthermore, the system must consider the inheritance hierarchy and possibly dynamic method dispatching. For example, a method `y()` added in some class `S` may be overridden in some subclass `T`. Both changes must be added to the same updating aspect, as the dynamic target type of method call of `y()` may either be `S` or `T`. Section 5 provides a detailed example.

## 4. ASPECT EXTRACTION

In this section, we describe the algorithms to extract the aspects from the differences between two complete versions of an application. To determine the updating aspects, the compile-time system analyzes the bodies of all changed methods. The system resolves all method invocations and then inspects whether the possible targets of the method call have changed. The system distinguishes between changes of the implementation and the specification, as the latter constitute dependences which must be encapsulated in the same aspect. In the following, we explain two algorithms which compute the aspects and the update dependences based on a method call graph including all potential method calls, taking into account static and dynamic target types.

Algorithm 1 describes the preparation and initialization for the updating aspects: The algorithm first identifies all methods that had changed only their implementation; the



```

public class AddedArgument extends DefaultAspect {
    /* Declare S.x(int) */
    public static int x(S s, int n) {
        return 42-n;
    }
    private Crosscut c = new MethodRedefineCut() {
        /* Match C.*(S) and redefine body */
        public void METHOD_ARGS(C target, S s) {
            print(x(s, 10));
        }
        /* Match C.m(*) */
        protected PointCutter pointCutter() {
            return Within.subType(C.class).AND(Within.method("m"));
        }
    };
}

```

**Figure 3: Change of the specification: AspectJ aspect in upper right, PROSE aspect below.**

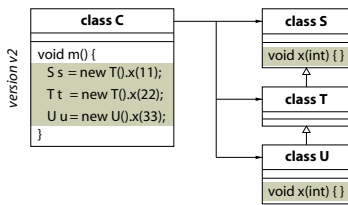
method body (lines 1-8). We call these methods *update roots* as they serve as starting point to determine the changes that must be encapsulated in the same aspect. For each update root, we declare an updating aspect to replace the former method body (lines 10-14). The number of implementation changes determines the upper bound of the number of aspects. As the updating roots may depend on the signature of other changed methods, the algorithm next calculates the dependences for each updating root (lines 16-18).

Algorithm 2 finds all methods that must be updated together with a given update root. To check for such dependences, the algorithm must first find all possible declarations of the methods the update root calls. Iterating over the instructions of the method body, the algorithm first *resolves* each method invocation (line 3). Function *getTargetImpl* determines the class that actually defines the method, starting from the class representing the static type of the call target. Next, the algorithm checks whether the resolved method has changed its specification (line 4). In this case, the changes of the caller and the callee must be updated together and the callee is added to the updating aspect (line 5). The algorithm recursively processes the method calls of all considered callee's (line 6). If the method is overridden in subclasses (in the case of virtual method calls), the redefined methods are added to the aspect (lines 7-10) and their dependences are analyzed as well (line 11).

Once all dependences and aspects have been identified, overlapping aspects (e.g., different callers depend on the same callee) are collapsed to reduce the number of updates. In Figure 4(a), both methods `c` and `h` depend on method `d`. The originally separate aspects `(c,d)` and `(d,h)` are merged into the aspect `(c,d,h)`. For space reasons, we do not provide the corresponding algorithm.

Finally, the compile-time system determines the aspect weaving order. We construct a directed acyclic graph (DAG) based on the call graph and the aspects (see Figure 4(a)).





```

public class Dispatching extends DefaultAspect {
    public static void x(S s, int n) { /* ... */ }
    public static void x(U u, int n) { /* ... */ }
    public static void dispatch_x(S s, int n) {
        if (s instanceof U) {
            x((U)s, n);
        } else {
            x((S)s, n);
        }
    }
}

public Crosscut c = new MethodRedefineCut() {
    public void METHOD_ARGS(C target) {
        S s = new T();
        dispatch_x(s, 11);
        T t = new T();
        dispatch_x(t, 22);
        U u = new U();
        dispatch_x(t, 33);
    }
    protected PointCutter pointCutter() {
        return Within.type(C.class).AND(Within.method("m"));
    }
};

```

Figure 5: Inheritance and method dispatching.

and to forward the method call accordingly. Figure 5 illustrates the inheritance hierarchy of the new version. Method `x(int)` was added to class `S` and overridden in class `U`. The aspect shows the added methods and the necessary dispatching method. Note that when updates are scattered across several classes, manual dispatching may also be necessary for all client classes.

### Approach limitations.

The approach to exploit aspects as primary updating unit allows us to express a variety of updates, such as addition or modification of methods. Besides to changes to the method body, we support changes to argument and return types as well as access modifiers. As removed methods cannot be physically removed from the run-time system, a `NoSuchMethod`-exception may be raised [5]. The change is reflected in the calling methods, and the removed method cannot be called any longer.

The approach cannot handle arbitrary object evolution, as an aspect system does not have the means to modify all existing objects to conform to a different class representation. We described a possible solution that employs a copying garbage-collector to traverse the object graph and to transform the objects [6]. Some object evolution scenarios can be accomplished by updating aspects. As aspects may specify actions to be executed on field read or write, type conversions may be realized on-the-fly. Other object changes are reflected in the caller methods (e.g., access level, initial value) and thus may be expressed by aspects. Aspects cannot deal with modifications of the inheritance structure and thus changes of the super-class or interfaces are not feasible.

## 6. CONCLUDING REMARKS

We have presented an approach to generate aspects for a dynamic software evolution system. We have shown that aspects are not only an interesting technique to represent crosscutting design concerns, but also have a more general applicability. As aspects provide a mechanism to specify exactly which code must be modified or added, we can use aspects to encapsulate changes that are scattered across several classes and that need to be updated together.

## 7. REFERENCES

- [1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, 1:135–173, 2006.
- [2] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. In *21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, 2006.
- [3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, 2004.
- [4] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniessel. Towards a Taxonomy of Software Change. *Journal of Software Maintenance*, 17(5):309–332, 2005.
- [5] S. Cech Previtali. Dynamic Updates: Another Middleware Service? In *1st Workshop on Middleware-Application Interaction (MAI'07)*, pages 49–54, 2007.
- [6] S. Cech Previtali and T. R. Gross. Dynamic Updating of Software Systems Based on Aspects. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 83–92, 2006.
- [7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.
- [8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [9] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05)*, pages 125–138, 2005.
- [10] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys'08)*, 2008.
- [11] A. Popovici, G. Alonso, and T. Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, 2003.
- [12] A. Popovici, T. R. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, 2002.
- [13] K. Waldén and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*. Prentice Hall, 1994. Available at <http://www.bon-method.com/>.
- [14] AspectJ website. <http://www.eclipse.org/aspectj/>.
- [15] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE), <http://java.sun.com/j2se/>.