Annotations for Seamless Aspect-Based Software Evolution

Susanne Cech Previtali and Thomas R. Gross

Department of Computer Science, ETH Zurich, Switzerland

Abstract. We are developing a dynamic software evolution system that leverages aspect technology to encapsulate software updates. Ideally, an evolution system provides as much automation as possible. Certain changes, however, defeat automation. For instance, field additions cannot be concisely captured without the feedback of the programmer. Rather than reconstructing the missing information in retrospect, we propose to gather the necessary meta-data along with the development process. We use Java annotations for that purpose: for instance, programmers may annotate added fields with their corresponding initialization. In some cases, the software development environment may even infer annotations from the actions taken by the programmer, and therefore, annotations enable a seamless software evolution cycle.

1 Introduction

Our approach to the dynamic evolution of object-oriented software systems [3, 4] treats updates in a manner similar to crosscutting concerns in aspect-oriented programming: all changes that belong to a logical update are encapsulated in one aspect. We are developing a software evolution system that implements this idea. To compute the required updates, the system compares *statically* two versions of a Java program in bytecode form and deduces their structural differences. The structural differences constitute the individual changes. The system identifies the dependences between the changes and encapsulates these changes as aspects. For deploying the aspects, the system relies on a dynamic aspect sub-system [7–10].

The software evolution system automates the version comparison, the deduction of update dependences, and the generation of aspects. The calculation of dependences and aspects depends on the extracted differences. Unfortunately, not all differences can be easily detected by a tool; e.g., name changes require special heuristics [5] or human interaction [2]. Furthermore, adding or changing the type of fields requires the programmer to provide code for transforming existing objects.

In this paper, we discuss how the programmer may convey meta-data in the form of Java annotations during software development and maintenance to automate the program comparison for update generation. The remainder of the paper is organized as follows: Sect. 2 discusses related work. Sect. 3 introduces annotations for our software evolution system. Sect. 4 concludes the paper.

2 Related work

The dynamic software updating (DSU) [5] system detects name changes of functions by partially matching the abstract syntax tree of successive versions of C programs. JDiff [1, 2] is a tool to compare object-oriented programs. The tool analyzes the bytecode from class files and compares programs at a statement-level granularity. To detect name changes, the authors propose human interaction during the process of comparison. Currently, our implementation does not detect name changes and consequently handles a rename as an addition and delete.

The DSU system [6, 12] generates type transformer functions, which must be completed by the developer to provide explicit conversions. Type evolution is handled by wrapping the original type and adding a version number and fixed-size extra space to prepare for eventual growth of the type over time. For each field access, the compiler inserts code to return the underlying representation. As type evolution is limited by the fixed amount of extra space reserved in the initial version, the authors mention the use of indirection in type definitions at the cost of an extra dereference per access. We suggest also to use indirection to accommodate object evolution, but, rather than providing type transformations when the release is ready, programmers encode the transformations along with the development, once they are aware of the details.

Robbes et al. [11] propose to record semantic changes from the refactoring information provided by an software development environment such as Eclipse to understand software evolution. Instead, we use refactoring information to generate Java annotations to achieve dynamic software evolution.

3 Annotations

In the following, we present how Java annotations may be used to tag changes and how the software evolution system can exploit this information.

3.1 Name changes

We propose a simple meta-data based approach to handle name changes. The approach relies on annotations that are generated by the software development environment whenever the programmer uses the renaming refactoring facility. The annotation must indicate the previous name of the renamed entity. Figure 1 shows two versions of a class. The new version (on the right) renames the method `start()` to `run()`; the annotation above the method declaration denotes the old name.

Currently, our implementation handles a rename as an addition and delete. With the information from the annotations, we can link these pairs and leave it up to the system to take proper action. The system may replace the new names with the original ones in the compiled class files or may update the symbol table.

```
/* Version 1 */
class S {

   void start() { ... }
}
```

```
/* Version 2 */
class S {
   @Rename (previous="start")
   void run() { ... }
}
```

Fig. 1. Annotations for renaming.

3.2 Field changes

Annotations may also facilitate field modifications that require object evolution, i.e., the adaptation of existing objects at run-time. Such changes modify the old object layout, by e.g., adding another field or by changing the type of an existing field.

Field additions. Adding a new field to a type requires all existing objects to be modified to support reading and writing this new field. Following Neamtiu et al. [6], we propose an indirection system in which a special field, ext, is added to all classes to support future growth. When a class must be updated with a new field, a special "delta class" is generated containing any added fields. Existing objects have their ext field set to an instance of this delta class, and field access in the original code is rewritten to accomodate the extra layer of indirection. Although the constructor in a new version initializes new objects appropriately (i.e., including the added field), special care must be taken to "initialize" the added field in existing objects. Using annotations, a programmer can indicate a method to initialize the added field.

Figure 2 provides an example. The new class declaration in the middle shows the added field field3 annotated with the name of the initialization method. This initialization method must be declared in the new class itself. The class declaration on the left contains a field ext of type Object to accommodate future field additions. The class declaration on the right lists the delta class; its special constructor links the existing object to this extension.

```
/* Version 1 */
class S {
   A field1;
   B field2;
   Object ext;
}
```

```
/* Version 2 */
class S {
   A field1;
   B field2;
   @Initialize(name="init")
   C field3;

   void init() {
      field3 = field1.addTo(field2);
   }
}
```

```
/* Update from version 1 to version 2 */
class DeltaS {
   C field3;
   Object ext;

   /* Special constructor */
   DeltaS(S old) {
      field3 = old.field1.addTo(old.field2);
      old.ext = this;
   }
}
```

Fig. 2. Annotations for field additions.

Type transformations. Another update concerns the modification of the type of an existing field. We propose two kinds of strategies to accommodate these changes. One strategy encompasses bidirectional type transformations. A bidirectional type transformation allows converting back and forth between two versions of a type. Examples are String changed to StringBuffer and int to Integer. The other strategy encompasses unidirectional type transformation. In contrast to bidirectional type transformation, unidirectional type transformations are not reversible. Examples are LinkedList changed to HashMap and int to float.

Both bidirectional and unidirectional type transformations may be expressed as annotations in the new version of the class. The annotation is attached to the corresponding field and indicates the necessary type transformation method. The example in Fig. 3 shows the new version of class S (on the left), which changed the type of its field from String to StringBuffer. The corresponding updating aspect (on the right) redefines a caller method C.run() to redirect the access using these transformation methods oldToNew() and newToOld() defined in the delta class DeltaS (not shown).

```
/* Version 2 */
class S {
  @BidirectionalTransformation
  (toNew="oldToNew", toOld="newToOld")
  StringBuffer msg;

  StringBuffer oldToNew(String prev) {
    return new StringBuffer(prev);
  }
  String newToOld(StringBuffer next) {
    return next.toString();
  }
}
```

```
/* Update from version 1 to version 2 */
class Update extends Aspect {
  /* Setters/getters of S, version 2 */
  void setMessage(S s, StringBuffer msg) {
    s.msg = DeltaS.newToOld(msg);
  }
  StringBuffer getMessage(S s) {
    return DeltaS.oldToNew(s.msg);
  }
  /* Redefine client using class S */
  redefine C.run() {
    S s = new S();
    setMessage(s, new StringBuffer("hallo"));
    StringBuffer msg = getMessage(s);
  }
}
```

Fig. 3. Annotations for type transformations.

Unidirectional type transformations may be handled similar to field additions. The field of the new type is added to the existing object and the type transformation function is used to initialize the added field. Consider a LinkedList that is replaced by an array; the elements of the array can be filled with the elements of the LinkedList. Field access must be reflected in the methods that use the fields. These methods will be redefined by updating aspects.

4 Concluding remarks

We have shown how meta-data in the form of Java annotations may facilitate software evolution. Such annotations are created along with the development

process and thus allow the programmer to indicate transformations while writing the new version. Annotations can either be directly provided by the programmer or inferred by the software development system. In the latter case, the software development system considers actions taken by the programmer and formulates an appropriate annotation. By including annotations and type transformations, the software evolution system may automate program comparison and update generation, and thus achieves a seamless evolution step.

Acknowledgments. 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.

References

1. T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 2–13, 2004.
2. T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Journal of Automated Software Engineering*, 14(1):3–36, 2007.
3. 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.
4. S. Cech Previtali and T. R. Gross. Extracting Updating Aspects from Version Differences. In *4th International Linking Aspect Technology and Evolution Workshop (LATE'08)*, 2008. *As the proceedings are not yet published, the paper is accessible at http://www.lst.inf.ethz.ch/research/publications/LATE_2008.html.*
5. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.
6. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, pages 72–83, 2006.
7. A. Nicoară and G. Alonso. Dynamic AOP with PROSE. In *International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05)*, pages 125–138, 2005.
8. A. Nicoară, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys'08)*, 2008.
9. 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.
10. 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.
11. R. Robbes and M. Lanza. A Change-based Approach to Software Evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, 2007.
12. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(4), 2007.