Representing Change by Aspect[*]

Peter Dolog, Valentino Vranić and Mária Bieliková
Dept. of Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology
Ilkovičova 3, 812 19 Bratislava, Slovakia,
{dolog,vranic,bielik}@elf.stuba.sk
http://www.dcs.elf.stuba.sk/~{dologp,vranic,bielik}

Abstract. We propose the application of aspect-oriented programming to software configuration management. We believe it could improve the change control by providing a new basis for reasoning about a change. To demonstrate this, we designed an abstract-oriented extension to procedural languages where a change is represented by an aspect. Consequently, a change gains the properties of an aspect: it becomes well-localized and separated from the (unchanged) base program. This goes beyond the current capabilities of configuration management methods and tools: the aspect representing the change can be applied to other versions of the program (possibly to different programs).

Keywords: aspect-oriented programming, change control, change representation.

1 Introduction

Software systems are developed and evolved in a series of changes. Changes arise as requirements are extended, reformulated, dropped or corrected, as faults are discovered, and in many other situations. The change is often required also due to a need for adapting the product to the user's context. We are witnesses of growing cooperation among software development companies. Many (often distributed) teams work on the same release of the software system in parallel. In such a situation, change control becomes even more important.

The level of change control support provided by the existing software configuration management tools varies significantly. Hence, if two companies decide to cooperate, there is a big chance that they would have different tools that provide different repository items representation, different structure representation, etc. The companies can also have different configuration management process established, including, for example, different branching and merging strategy, what even more complicates keeping track of changes in the source code.

We propose a solution to some of these problems by treating a change at the source code level and by expressing it explicitly. To achieve this, we employ the aspect-oriented programming, a new approach to programming aiming at separation of crosscutting concerns (see Section 3).

The rest of the paper is structured as follows. First, we put our approach in the context of the existing versioning models (Section 2) and the aspect-oriented programming (Section 3). Then we present an abstract aspect-oriented extension to procedural languages (Section 4). Subsequently, we show how this extension can be concretized to VBScript language (Section 5). Finally, we draw some conclusions and point some directions for the further work (Section 6).

2 Version Models and Change

A version model defines the entities to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions [4]. Several version models are described in the literature and used in existing configuration management tools. We focus on the core issue of versioning, namely the organization of version space, or to be more specific—the version description and representation. Our main interest is to improve change control.

According to the entities being handled, the version models are classified into state-based and change-based. *State-based models* focus on the states of versioned items. In such approach, versions are usually described in terms of revisions and variants [2]. A configuration item (the smallest unit of a system taken under version control) is maintained usually at the file level. The change in state-

[*]This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20.

based models can be described as the difference between two versions. Many commercial systems are state-based (e.g., Microsoft Visual Source Safe, Rational ClearCase, PVCS) [3].

The problem with state-based models is that a change is maintained implicitly, during the modification of a branch. Merging can be viewed as a re-application of all the changes to the branches being merged. This requires the extraction of the changes from the branches and their subsequent application to the base.

In *change-based models* the change is treated as a first class entity and managed explicitly by a developer, either manually or by a tool. A version is considered as the result of the application of changes to a baseline. There are several commercial change-based software configuration management systems that have the ability to track the logical changes rather than individual file changes (e.g., Continuus/CM, CCC/Harvest). They treat the change at the level of the source code lines or at the level of the file versions. Accordingly, they allow to create change sets or change packages [13]. A *change set* consists of the changed code lines. A *change package* contains references to the file versions that are the compositions of logical changes.

In a change-set model, changes are combined freely to construct new versions according to the requirements. In [4] such approach is denoted as *change-based intensional versioning*. The use of change packages is denoted as *change-based extensional versioning*, because version set is defined explicitly by enumerating its members. In this case, each version is described by changes relative to some baseline.

Another change-based approach is based on change identification by language constructs. This can be denoted as *language-aware* approach: the change is handled by directives for source code inserting, deleting and editing augmented with the attributes of the change (e.g., who and when made the change, etc.). An example of this approach is VTML (Versioned Text Markup Language) [9] or conditional compilation.

The conditional compilation enables to use the preprocessor directives to control the code fragments visibilities. In this case, all changes (fragments) are stored in one file, which is hard to maintain. Management of fragments' visibilities is necessary for improving change control. This approach is used, for example, in the EPOS system [5].

The change-based systems are not so widely used as the state-based ones. The main reason is that developers think rather at the version-state level. However, nowadays many state-based systems are being extended to provide the change-based functionality (e.g., Rational ClearCase) [8]. The objective is to improve change management and traceability of the change request in a software development process.

The change representation influences the change control procedure, which consists of the four major steps [1]: checking whether the change is needed,[1] analysis of causes that led to change, planning the change, and change implementation. In the context of the change control procedure, we are concerned with the change implementation.

The problem with the surveyed change-based approaches is the granularity of the logical change. As we mentioned, some of them treat the logical change as the individual lines of the source code, while other are based on representing change by preprocessor directives.

3 Aspect-Oriented Extensions

The main idea of the aspect-oriented programming (AOP)—separation of concerns by separating the crosscutting concerns called *aspects* from the basic functionality crosscut by them—is carried throughout several independently developed approaches [10, 12]. Among them, Xerox PARC AOP [14] holds a significant position. Further in the text by the AOP we mean actually the Xerox PARC AOP.

AOP appeared as a reaction to the problem known from the *generalized procedure languages* [7], i.e. programming languages that use the concept of the procedure to capture the functionality.[2] In such languages the program code fragments that implement a clearly separable aspect of a system (such as synchronization) are scattered and repeated throughout the overall program code that, in advance, becomes *tangled*. AOP aims at factoring out such aspects into separate program units called by the same name: *aspects*. Aspects *crosscut* the *base* code in places called *join points*. These must be specified so aspects could be *woven* into the base code by the program called *weaver*.

The join points can be static or dynamic. *Static join points* can be identified in the program text itself. They can be specified in terms of a programing language syntax alone. An example of such a join point is the beginning or end of a method or procedure body. *Dynamic join points* are available at run time only. For example, a method reception by an object is a dynamic join point. In the weaving process, the static join points are resolved by a simple program code insertion, while dynamic join points can be resolved at run time only.

The special language constructs used to capture the aspects and join points are known as the *aspect-oriented extension* of the base language. The two types of aspect-oriented extension regarding its relationship to the base

[1]It is possible that some workaround for the existing activity could be more effective than the change itself.

[2]Besides the procedural languages, these include functional and object-oriented languages as well.

language can be distinguished: homogenous and heterogeneous. The homogenous extension, besides for some additional constructs, relies on the base language to the greatest possible extent, while the heterogeneous extension introduces a whole new language for capturing the aspect-oriented part of the program. In general, there can be several independent aspect-oriented extensions, handled by the same or by separate weavers.

Not unlike programming languages in general, an aspect-oriented extension (including the corresponding weaver) can be designed to solve a specific problem, such as the one presented in [7] (the filtering example), or to serve a general-purpose, as the AspectJ language [15], which is a homogenous, general-purpose aspect-oriented extension to Java. While aspect-oriented extensions provide a new way of programming, they do so only in the context of the language they extend. In other words, AOP is a multi-paradigm approach in its very nature [12], and AspectJ can be viewed as a multi-paradigm language [11].

4 Aspect-Oriented Extension for Change Representation

As it was discussed in Section 2, current configuration management approaches do not offer a satisfactory change representation regarding the change maintenance and re-applicability to different branches. The use of AOP enables to maintain changes explicitly by capturing a change into an aspect.

In order to enable change representation by aspect, the aspect-oriented extension to a given programming language should be provided. Since the changes are actually changes of the program text, all the join points will be statical. Further, the aspect-oriented extension should be *homogenous*—to preserve the base language constructs, and *general-purpose*—to cover all the types of changes (which depend on the base language). Also, the join point description should not affect the base program.

To illustrate aspect-oriented approach to change representation, we developed an aspect-oriented extension (inspired by AspectJ) to procedural languages. Proposed language constructs are presented in Fig. 1. Different type styles are used to distinguish among the `keywords`, **required parts** and *optional parts*.

The aspects are placed into modules, possibly together with the ordinary procedures which can be called from within the aspects, i.e. inside of the *block* parts. The *block* parts must be parsed either by the weaver, or by the original language parser.

The introductions are used to introduce new procedures and variables into modules (M_i). The advices enable performing a command *block* before, after or in place of the procedures determined by a specified set of join points, so-called *pointcut*. While `before` and `after` advices are simple, the `around` advice requires some explanation. It enables to run an initial block $block_i$, then to `proceed` with the next action, which is either another aspect, or the original procedure body, in case there is no other aspect affecting the procedure. The optional *return_clause* in `after` and `around` advices enables to modify the return value (if the procedure returns one) before it is actually returned to the caller.

The **pointcut_specification** is built out of the pointcut primitives (listed in the bottom of Fig. 1) using the logical operators *and* and *or*.[3] The parentheses can be used to declare the priority of subexpressions evaluation. The first two primitive pointcuts, `modules` and `withincode`, designate all the join points within the modules M_i and procedures specified by the **procedure_signature**, respectively. The `calls` pointcut primitive designates all the procedure calls specified by the **procedure_signature**. The `definitions` pointcut primitive designates the actual definitions, i.e. bodies of the procedures specified by the **procedure_signature** (see Fig. 5 for an example). A `before` advice to a `definitions` pointcut will insert its code *after* all the declarations of variables in the specified procedure(s) placed before the first non-declaration statement.

The wild cards `*` and `..` can be used in **procedure_signature** to denote any string of characters and omitted arguments, respectively. This convention is used in AspectJ, e.g. `* p*(int, *)` denotes all the methods whose name starts with `p`, with one `int` argument and one argument of any type, returning a value of any type. The most general signature—denoting all the methods—is then `* *(..)`.

Up to now we said nothing about the optional *argument_list* in advices. It is used to access the arguments of the procedures denoted by the pointcut. Suppose we want to make a `before` advice to the following C function:

```
int f(int i) {return i*i;}
```

Consider these two advices:

1. `before(): definitions(int f(int)) {i = i + 1;}`

2.
   ```
   before(int x): definitions(int f(x))
         {x = x + 1;}
   ```

Both advices seem to do the same thing; they add one to `f`'s argument before proceeding with the rest of `f`'s body.[4] However, if we rename `i` in function `f` to `j`, the first advice will fail to satisfy our intention (moreover, it will produce a syntax error), so the second version is obviously more robust.

[3]Since pointcuts are the sets of join points, the *and* and *or* operators have the meaning of set intersection and union, respectively.

[4]This is different from AspectJ where the method body is not visible to advices.

Introductions:

`introduction` $\mathbf{M}_1$`,...,`$\mathbf{M}_n$ {*block*}

Advices:

`before(`*argument_list*`):` **pointcut** {*block*}

`after(`*argument_list*`):` **pointcut** {*block return_clause*}

`around(`*argument_list*`):` **pointcut** {$block_i$ `proceed...`$block_f$ *return_clause*}

Pointcuts:

`pointcut` **pointcutName** (*argument_list*): **pointcut_specification**

Pointcut primitives:

`modules(`$\mathbf{M}_1$`,...,`$\mathbf{M}_n$`)`

`withincode(`**procedure_signature**`)`

`calls(`**procedure_signature**`)`

`definitions(`**procedure_signature**`)`

Figure 1: Aspect-oriented extension to procedural languages.

The proposed aspect-oriented extension is capable of describing the following types of changes:

- introduction of a new procedure or (global) variable into the module;
- extension of a procedure by a code before, after, or instead of it;
- change to the procedure arguments and return value.

What all of these changes have in common is that they are all about the functionality. The changes that cannot be described at the level of functionality are very hard (or impossible) to deal with using the aspect-oriented approach. These include renaming a procedure or variable, adding a white space or comment, changing the position of a procedure in the source code, etc.

A version is obtained by weaving the aspects that capture the change into the base program. Since this version might become a subject of modification as well, it should be human readable. This is different from the AOP itself where the process of weaving yields only an intermediate product not intended to be read by a human. The aspect-oriented extension proposed principally satisfies this requirement, since it relies on static join points only.

5 Case Study: Script Customization

We will show now how the approach we proposed in the preceding section could help in solving a problem of synchronizing the local customization with the global version of a program in script languages by the means of an example. We will use VBScript-like syntax since VBscript is widely used as a language for dynamical content generation and design of the web pages. It is the core language for Microsoft's `asp` pages. Further, some software houses use VBScript as the language for customizing their products, e.g. InteractCommerce corp.'s SalesLogix.

Suppose that two teams work on one system. The teams received change requests regarding the same script, shown in Fig. 2, which is a part of the system, at the same time, i.e. before synchronization of branches, as depicted in Fig. 3. The purpose of the script is to extract the list of sales opportunities from the `opportunity` table in the `test` database. The change request received by the first team is about extending the list of opportunities by the list of products. A new recordset, as well as the SQL statement and several lines of VBScript code must be added in order to accomplish the task of extracting the records from the table and generating the page containing the data.

The modified script is presented in Fig. 4; some commands the same as in Fig. 2 have been omitted (indicated by ellipsis). The code between the `change` and `end change` comments can be separated into the aspect module, as presented in Fig. 5. The affected module is specified by the `modules` designator in both advices. The declarations of additional variables are provided in the `before` advice. The conjunction of the `definitions` and `modules` designator states that the sequence of variable declarations in the advice is to be merged *after* all the declarations in the `main` procedure which are placed before the first non-declaration statement. The sequence of the directives to be run after the `ro.close` method in-

```
sub main
  Dim con 'Connection object
  Dim s 'Select statement
  Dim ro 'Recordset object
  Set con = Server.CreateObject("ADODB.Connection")
  con.Open "Test"
  s = "SELECT * FROM opportunity"
  Set ro = con.Execute(s)
  call gener_data(ro)
  ro.Close
  con.Close
  Set ro = Nothing
  Set con = Nothing
end sub
```

Figure 2: The code base in VBScript.

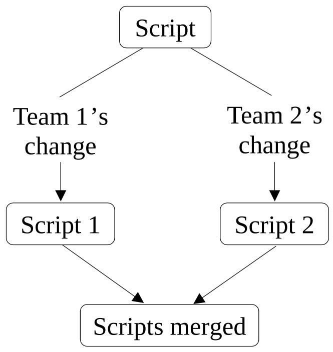

Figure 3: The branching.

vocation (specified by the `calls` designator) within the original `main` procedure (specified by the `withincode` designator) has been enclosed into the `after` advice. The result of the merging, i.e. weaving, will be the same code as displayed in Fig. 4.

The other change request addressed to the second team resulted in the script shown in Fig. 6. The second team's change consists of adding the `while` loop for updating the `applied` and the `date_of_application` fields for each record in the opportunity table, and of adding the sequence of commands that generate the list of marketing campaigns from the `marketing` table. We can apply the aspect from Fig. 5 to the code in Fig. 6 without any change.

However, that change could have been separated into the aspect, too, if both teams used the aspect-oriented approach. In that case, we would simply apply the two aspects subsequently in order to obtain both functionalities.[5]

[5]The priority of aspects is not significant here, but this is not so in general.

```
sub main
  . . .
  '***change of declarations***
  Dim rp 'Recordset object
  Dim s2 'Select statement
  Dim c 'Command object
  '***end of change***
  Set con =  . . .
  . . .
  ro.Close
  '***change***
  s2 = "SELECT * FROM product"
  Set c = Server.CreateObject("ADODB.Command")
  c.ActiveConnection = con
  c.CommandText = s2
  Set rp = c.Execute
  call gener_data(rp)
  rp.close
  rp = Nothing
  c = Nothing
  '***end of change***
  . . .
end sub
```

Figure 4: The change performed by the first team.

6 Conclusions

We proposed a new approach to change-oriented versioning based on the aspect-oriented programming. The contribution of this paper is the proposal of the technique aimed to simplify change control by reifying the changes into language-level entities: a change is represented by an aspect and maintained explicitly by a developer.

A homogenous, general aspect-oriented extension has to be provided for a given programming language first. For the purposes of our approach, it is sufficient if this extension supports static join points. Since procedural, functional and object-oriented languages are easily extended to support the AOP with static join points, this approach is low-cost. We proposed such an extension to procedural languages. Moreover, it can be expected that general aspect-oriented extensions to other programming languages will be developed and provided for the sake of the AOP itself, so no additional effort would be necessary to employ this approach in such languages. This can be denoted as *self-supported change management*: a change is represented by the constructs that are a part of the programming language itself.

We assume this as one of the main advantages of the proposed version space representation. It provides a new base level for the change control; it is a move from the change control at the line level to the one at the programming language semantics level. In small software projects it is directly usable even without a software configuration management tool. The change comprehension and ori-

```
before(): modules(script) && definitions(main)
  begin
  Dim rp 'Recordset object
  Dim s2 'Select statement
  Dim c 'Command object
end before

after(): modules(script) && withincode(main)
         && calls(ro.close)
begin
  s2 = "SELECT * FROM opportunity_product"
  Set c = Server.CreateObject("ADODB.Command")
  c.ActiveConnection = con
  c.CommandText = s2
  Set rp = c.Execute
  call gener_data(rp)
  rp.close
  rp = Nothing
  c = Nothing
end after
```

Figure 5: The change separated into the aspect.

entation in the source code is easier because the change is well-localized in the aspect and need not be searched for. A change is possibly re-applicable as is or with some adaptation of the aspect involved (white-box reuse). Actually, the aspect can be applied to a completely different module than it was intended for by a simple modification of the pointcut.

Aspect-oriented approach can be used also in post-deployment configuration management [6] for parametrization (modification of a software system to take into account the local site context). The local context can be represented by an aspect. The application of the relevant aspects provides customization of the new product version according to the local context (developed for the previous version). Obviously, new aspects will be also created, in order to customize new features in the current version of the product.

Our approach can be used with existing software configuration management tools. Moreover, our approach is independent of the model employed by software configuration management tools. An aspect is a separate item, so it can be handled in both basic version models (state-based and change-based). It also supports the implicit long transaction maintenance because aspect itself represents a change and it is up to the developer to decide when the change should be committed. As aspects can be simply plugged in or out before the compilation, adding of an individual change into a version or substracting a change from a version (similarly as in the change-set approaches) is simple. Even more, the aspects can be combined into change packages. A change request could be then directly assigned to the corresponding aspect or change package

```
sub main
  . . .
  '***change of declarations***
  Dim rm 'Recordset object
  Dim com 'Command object
  Dim str 'String - select statement
  '***end of change declarations***
  Set con =  . . .
  . . .
  call gener_data(ro)
  '***change***
  While Not ro.EOF
    ro.Fields("applied") = 'T'
    ro.Fields("date_of_application") = Now
  Loop
  str = "SELECT * FROM marketing"
  Set com = Server.CreateObject("ADODB.Command")
  com.ActiveConnection = con
  com.CommandText = str
  Set rm = com.Execute
  call gener_data(rm)
  rm.close
  rm = Nothing
  com = Nothing
  '***end of change ***
  ro.Close
  . . .
end sub
```

Figure 6: The change performed by the second team.

(indicated by the appropriate identifiers).

Our work is now oriented toward a deeper elaboration of practical use of the proposed approach. Some additional mechanisms should be added to manipulate version history and versions themselves. In order to be able to control the changes effectively, some meta-data should be stored within each change (e.g., who and when made the change). In order to follow a version history, the meta-data related to changes should be processed and interpreted.

On the other hand, we are already able to partially track the version history, but it is difficult to determine which version was created by a developer and which is just a potential version when storing only changes. The potential versions can be obtained by applying the combinations of the aspects. However, not all potential versions make sense [4].

An additional problem is that the aspect representing a change can become a subject of change, too. As a consequence, a method of dealing with the change of a change should be proposed. This problem arises in any change-based version model, of course, but a special method is needed here because our approach works at other level of change control than traditional change-based version models.

References

[1] George W. Allan. *An Holistic Model for Change Control*, pages 703–707. Plenum, New York, 1997. Available at `http://www.dis.port.ac.uk/~allangw/chng-man.htm`. Accessed on March 6, 2001.

[2] M. Bieliková and P. Návrat. An approach to automated building of software system configurations. *Int. Journal of Software Engineering and Knowledge Engineering*, 9(1):73–95, 1999.

[3] Jim Buffenbarger and Kirk Gruell. A branching/merging strategy for parallel software development. In Jacky Estublier, editor, *Proc. of 9th Int. Symposium on System Configuration Management*, pages 86–99, Tolouse, France, September 1999. Springer LNCS 1675.

[4] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[5] Bjorn Gulla, Even-André Karlsson, and Dashing Yeh. Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386, November 1991.

[6] Dennis Heimbigner and Alexander L. Wolf. Post-deployment configuration management. In Ian Sommerville, editor, *Proc. of 6th Int. Workshop on Software Configuration Management*, pages 272–276, Berlin, Germany, March 1996. Springer LNCS 1167.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vidiera Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. of 11th European Conf. on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, June 1997. Springer LNCS 1241. Available at [15].

[8] David B. Leblang. Managing the software development process with ClearGuide. In Reidar Conradi, editor, *Proc. of 7th Int. Workshop on Software Configuration Management*, pages 66–80, Boston, USA, May 1997. Springer LNCS 1235.

[9] Fabio Vitali and David G. Durand. Using versioning to support collaboration on the WWW. In *Proc. of 4th World Wide Web Conference*, 1995. Available at `http://www.w3.org/pub/Conferences/WWW4`. Accessed on March 6, 2001.

[10] Valentino Vranić. Multiple software development paradigms and multi-paradigm software development. In J. Zendulka, editor, *Proc. of the Information Systems Modelling 2000*, pages 191–196, Rožnov pod Radhoštěm, Czech Republic, May 2000. MARQ.

[11] Valentino Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In *Proc. of Third International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, Erfurt, Germany, September 2001. Springer. Accepted for publishing.

[12] Valentino Vranić. Towards multi-pradigm software development. Submitted to Journal of Computing and Information Technology (CIT), 2001.

[13] Darcy Wiborg Weber. Change sets versus change packages: Comparing implementation of change-based SCM. In Reidar Conradi, editor, *Proc. of 7th Int. Workshop on Software Configuration Management*, pages 25–35, Boston, USA, May 1997. Springer LNCS 1235.

[14] Xerox PARC. Aspect-Oriented Programming home page. `http://www.parc.xerox.com/aop`. Accessed on July 11, 2001.

[15] Xerox PARC. AspectJ home page. `http://aspectj.org`. Accessed on July 11, 2001.

Peter Dolog received his Bc. (BSc.) in 1998, and his Ing. (MSc.) in 2000, both in information technology, and both from Slovak University of Technology in Bratislava. Since 2000 he is a PhD student at the Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology of Slovak University of Technology in Bratislava. His research interests include hypermedia systems modelling, adaptive presentation of information in the Internet, and new approaches to software engineering in general. He is a member of the Slovak Society for Computer Science.

Valentino Vranić received his Bc. (BSc.) in 1997, and his Ing. (MSc.) in 1999, both in information technology, and both from Slovak University of Technology in Bratislava. Since 1999 he is a PhD student at the Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology of Slovak University of Technology in Bratislava. His main research interests are multi-paradigm software development and aspect-oriented programming. He is a member of the Slovak Society for Computer Science.

Mária Bieliková received her Ing. (MSc.) in 1989 from Slovak University of Technology in Bratislava, and her CSc. (PhD.) degree in 1995 from the same university. Since 1998, she is an associate professor at the Department of Computer Science and Engineering at Slovak University of Technology. Her research interests include knowledge software engineering, software development and management of versions and software configurations, adaptive hypermedia and educational systems. She is a member of the Slovak Society for Computer Science, IEE, ACM, IEEE and its Computer Society.