A Version Model for Aspect Dependency Management

E. Pulvermüller[1] A. Speck[2] J. O. Coplien[3]

[1]Institute for Program Structures and Data Organization
Universität Karlsruhe, Germany
http://i44w3.info.uni-karlsruhe.de/~pulvermu

[2]Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, Germany
http://www-pu.informatik.uni-tuebingen.de/users/speck

[3]Bell Laboratories Lucent,
Naperville IL, USA
http://www.bell-labs.com/~cope/

Abstract

One observed characteristic of AOP is that it results in a large number of additional (coarse-grained to fine-grained) system units (aspects, concerns) ready to be composed to the final application. With this growing number of system units the dependencies between them become vast and tangling. Our paper investigates this problem, proposes a more general model (version model) to capture different facets of AOP as well as a partial solution towards unit consistency based on versions.

1 Introduction and Problem

Aspect-oriented programming (AOP) extends the potential of common object-oriented software engineering. Besides classes and objects, aspects [9] or (more general) concerns are an additional type of system units.

Beyond the identification and definition of concerns a further problem arises: How can an aspect or concern configuration be verified and how can the correctness of their mutual dependencies and interactions be proved? The more concerns available the more complex and error-prone their combination if manually practiced. The importance of this problem in the context of aspect-oriented programming has already been detected in research as may be observed in [19, 15, 18].

Two examples of dependencies in practice are:

- Aspects in distributed systems [14]:
  In order to keep an application independent from the communication technique the communication code may be separated in aspects. When a client wants to access a service exposed by a specific server the client has to obtain an initial reference to the server. This can be done either via a name server or via a file-based solution (the reference is stored as a string in a file which is accessible for clients and server). Aspects realizing one of these two alternatives are exclusive. This is already known at design and implementation time.

- Cross-cuts in embedded systems [17]:
  In the growing market of small embedded systems the industry aims at reusing both control hardware (with small adaptations to perform the specific services) and the corresponding software. The same microcontroller with a core software may be used in digital I/O devices or analogue I/O devices as well as in incremental resolvers. Except to a few statements (handling the internal data-flow) the software may be the same for all these devices. These statements may be ideally realized as aspects. The problem here is to assert that an aspect set woven into a specific base code is complete and that there are no date-flow statements missing (especially with respect to security and error handling code).

In the remainder of the paper we introduce a version model and show how it captures and extends the idea of AOP and, moreover, how it provides a means to preserve consistency and correctness.

2 Version Model

Our approach for the aspect dependency problem is based on a broader viewpoint onto different granularity levels [1] of concerns: a version model. The version model integrates consistency and dependency management in a natural way. It allows the description of the internal structure of an aspect as well as the dependencies between aspects of a set.

A version model is a model explaining the construction of software systems using the notation of versions. A version describes a software core which may contain other versions and has to consist of a valid set of conditions.

[1] Entire architectures may contain tangling classes or methods, classes may be cross-cut by methods or attributes and methods may be intersected by single statements.

Definition: (*Version*)
The symbol reflecting a version is V_i^k where k represents the granularity (0 stands for the most low-level granularity) and i gives the index distinguishing between versions on the same level of granularity.

Now we can inductively define the construction of versions:

$$Version\ V_i^0 \ = \ 1 \ \wedge \ Cond_i^0$$

where $Cond_i^0$ represents the condition [2] that has to be true for one version V_i^0 on level 0. $V_i^0 \in V^0$ where

$$V^0 \ = \ \{\bigcup_{j=1}^{\infty} V_j^0\}$$

is the set of all versions on level 0.

Similarly we can define the induction step:

$$Version\ V_i^{(n+1)} \ = \ \bigwedge_{j=1}^{m} V_j^n \ \ \wedge \ \ Cond_i^{(n+1)}$$

with

$$Cond_i^{(n+1)}\ is\ true,\quad 1 \leq l \leq m \leq |V^n|,\quad V_j^n \in V^n$$

□

In other words: a version is a set of conditions (a unification of the particular conditions of a certain version and all conditions of the subversions contained in the version). A condition is expressed as boolean expression.

The operands in such an expression are conjunction, disjunction, negation. An example for a condition may be:

$$V2 \wedge (\neg C3 \vee C4) \wedge (V3 \wedge C3)$$

[2] Several conditions may be unified in one condition.

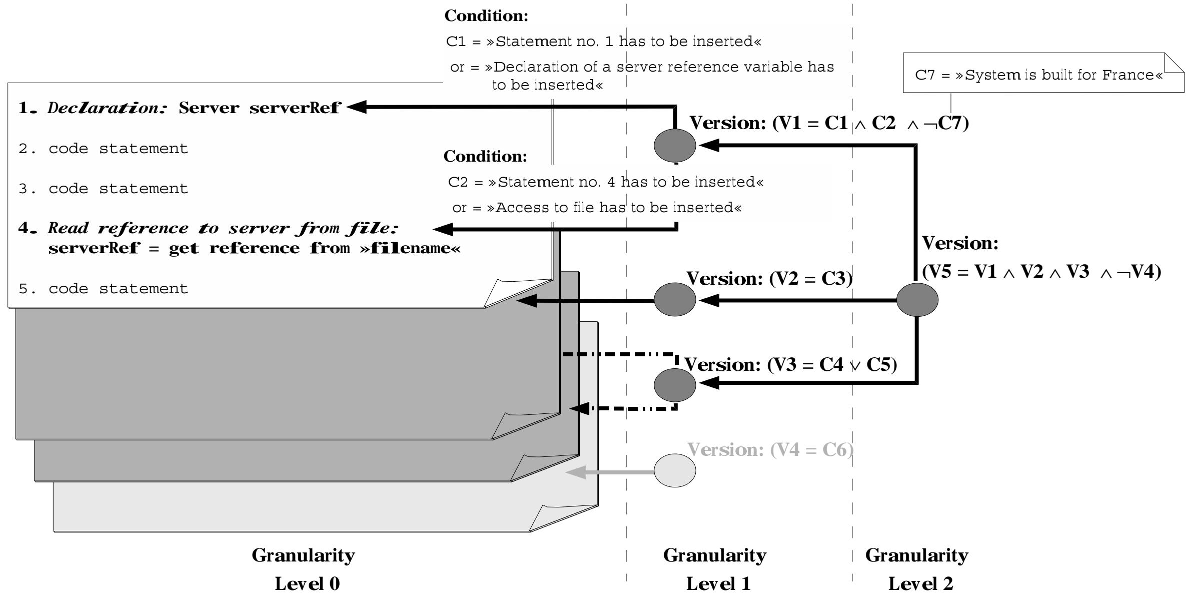

Figure 1: Versions in a Distributed System

where $C3$, $C4$ represent some defined conditions (e.g. $C3$ represents something like "This version is only reasonable if used in France since the dialogues are in French language.").

The condition Vn (in the example: $n = 2$ or $n = 3$) means that a version V_i^k for one possible k and i with name Vn is part of the (partial) system.

We now use this version model to model both individual aspects as well as sets of aspects forming a valid configuration (i.e. resulting in a valid and reasonable system when woven).

Some of the atomic conditions (in the first case), for instance, have the following appearance: $S1 =$ "Statement 1 (code line 1) has to be in the aspect".These atomic conditions filter (and thus structure) some part of the total code into aspects.

Atomic conditions in the second case (i.e. the modeling of valid aspect sets based on versions) are, for instance: $A1 =$ "Aspect with name $A1$ has to be part of the system".

The example in figure 1 depicts a version ($V5$ respectively V_5^2) for a distributed system built from individual aspects (cf. section 1 first problem). V_5^2 consists of V_1^1 and V_2^1 and V_3^1 but excludes V_4^1. In other words V_5^2 is valid when all the expressions or conditions of V_1^1, V_2^1 and V_3^1 are true and V_4^1 is false. Note that the versions of granularity level 1 also contain conditions which have to be true.

The version model, therefore, provides a consistent support for the software developer giving aid in producing semantic reasonable aspects and aspect combinations. It is independent of the underlying implementation technique (e.g. AspectJ [9] or meta-programming approaches [5, 12]) realizing the concrete aspects and software system.

The advantage of this formalization is that it provides a base for automatic configuration and checking. Having defined all relevant conditions it becomes possible to evaluate these boolean expressions and to find combinations which are semantically wrong.

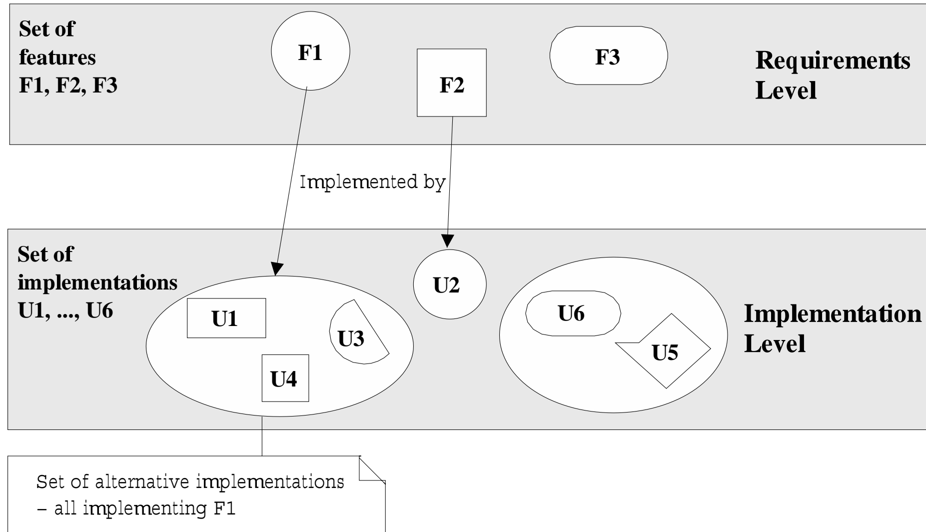

Figure 2: Relationship between Requirements and Implementation Level

3 Conditions

There are several issues to be considered with respect to conditions: The classification, detection, collection, storage and evaluation of conditions as well as the different ways to express conditions in a formal way (allowing automatic evaluation). Each is an area of research by itself and in this paper we only touch some of these issues.

The concrete conditions are application dependent which means that they appear during the problem analysis and design of a system. Storing the conditions, i.e. the versions, in a repository results in a better reuse [10]. Besides the reuse of concerns it becomes possible to use existing condition information to decide whether the reuse is semantically reasonable in a certain context.

Following the traditional model for software development we can distinguish between two levels of semantic knowledge: Conditions referring to the high-level requirements or to individual implementations. There is a clear connection between these two levels as exposed in figure 2.

The formal relationship is: $U1, U2 \in Set1$; $F1$, $F2 \in Set2$ where $Set1$ is the set of implementation units and $Set2$ is the set of features on the requirements level. Assuming that $U1$ implements (besides others) feature $F1$ and $U2$ implements $F2$ then:

$$valid(U1, U2) \Rightarrow valid(F1, F2)$$
$$valid(F1, F2) \not\Rightarrow valid(U1, U2)$$

with function

$$valid(X, Y) = \begin{cases} 1 & : \text{X and Y form a valid combination} \\ 0 & : \text{X and Y form an invalid combination} \end{cases}$$

Function $valid(X, Y)$ may be calculated by evaluating the binary condition expressions.

The distinction between requirements and implementation level is not only limited to the development phase of a system or concerns but also exists in the maintenance phase where additional conditions may appear. This is due to the fact that

it is impossible to capture all relevant dependencies and conditions from beginning. Additional conditions are added as needed or detected in a piecemeal growth manner [4].

Domain engineering [6] is a powerful means to detect and capture reoccurring and thus reusable conditions on the requirements level for a certain domain. While modeling the commonalities and differences of a domain, e.g. in a feature model [6], it is possible to extend this model by additional semantic information or even derive logical formulae directly from the model (the model already captures semantic relationships as feature interdependencies).

Until now we regarded the conditions as (binary) formulae being true or false. Though this is the first step for a mathematical foundation it does not yet reflect all the facets of the reality. An extended version model also considers values between 0 and 1, temporal and contextual information and dependencies.

4 Related Work

Related work may be found in all aspect-oriented and related approaches e.g. subject-oriented programming [8], adaptive programming [11], adaptive plug & plays [13], composition filters [1] or also transformation systems [3].

The proposed version model is orthogonal to these approaches as it provides a means to describe the system units in a more abstract way on different levels of granularity (in the form of a construction instructions). With the definition of versions it is possible to extract different views (i.e. versions) onto one unit. The most important difference is that the version model integrates (even focuses on) consistency and dependency management (thus addresses the aspect dependency problem [10]) in a natural way.

With respect to composition validation further work may be found in [2]. In the GenVoca model composition is described with type equations. A design rule checking mechanism detects illegal combinations. GenVoca is a layered model and thus only layered composition is considered.

Prior work about versioning at Bell Laboratories and in [16, 20], for instance, has influenced our version approach.

Generative programming [6] is another related field which may augment the version model. The version notation may serve as an input for generators. Feature models may be an important technique in this context.

AI technologies can help to manage semantic knowledge. [7] describes a way to use expert systems to support reuse of object-oriented frameworks by means of explicitly encoded design knowledge and user interaction. Concerns composition conditions are one type of such semantic design knowledge.

5 Conclusion

The version model proposed in this paper allows both, to describe the internal structure of an aspect and to define valid clusters of concerns. Using binary logical formulae for that purpose provides, moreover, a mathematical foundation to prove correctness of composition. This opens the field of logic and all its algorithms.

An extended version model also considers condition values in the range between 0 and 1, temporal and contextual information and dependencies.

References

[1] M. Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4), December 1996.

[2] D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Genera-

tors. In *IEEE Transactions on Software Engineering*, pages 67 – 82, 1997.

[3] I. Baxter. Design Maintenance Systems. *Communications of the ACM*, 35(4):73 – 89, April 1992.

[4] J.O. Coplien. Re-evaluating the Architectural Metaphor: Towards Piecemeal Growth. Guest editor introduction to IEEE Software Special Issue. *IEEE Software Special Issue on Architecture Design*, 16(5):40 – 44, September 1999.

[5] K. Czarnecki and U.W. Eisenecker. Synthesizing Objects. In *Proceedings of ECOOP'99, European Conference on Object-Oriented Programming*, LNCS 1628, pages 18 – 42. Springer-Verlag, June 1999.

[6] K. Czarnecki and U.W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] W.D. De Meuter, M. D'Hondt, S. Goderis, and T. D'Hondt. Reasoning with Design Knowledge for Interactively Supporting Framework Reuse. http://progwww.vub.ac.be/Research/ResearchPublicationsDetail2.asp?paperID=81, February 2001.

[8] Ossher H. and P. Tarr. Using Subject-Oriented Programming to overcome common Problems in Object-Oriented Software Development/Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 687 – 688, May 1999.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *LNCS 1241*, ECOOP. Springer-Verlag, June 1997.

[10] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the GCSE'00, Second International Symposium on Generative and Component-Based Software Engineering*, LNCS, Erfurt, Germany, September 2000. Springer. to appear.

[11] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[12] A. Ludwig and D. Heuzeroth. Metaprogramming in the Large. In *Proceedings of the GCSE'00, Second International Symposium on Generative and Component-Based Software Engineering*, LNCS, Erfurt, Germany, September 2000. Springer. to appear.

[13] M. Mezini and K.J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *ACM SIGPLAN notices*, volume 33, October 1998.

[14] E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in Distributed Environments. In K. Czarnecki and U. W. Eisenecker, editors, *Proceedings of the GCSE'99, First International Symposium on Generative and Component-Based Software Engineering*, LNCS 1799, Erfurt, Germany, September 2000. Springer.

[15] E. Pulvermüller, A. Speck, M. D'Hondt, W.D. De Meuter, and J.O. Coplien. Workshop on Feature Interaction in Composed Systems, ECOOP 2001. Budapest, Hungary, June 2001. To be held.

[16] M.J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364 – 370, December 1975.

[17] A. Speck, E. Pulvermüller, and M. Mezini. Reusability of Concerns. In C. V. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr, editors, *Proceedings of the Aspects and Dimensions of Concerns Workshop, ECOOP2000*, Sophia Antipolis and Cannes, France, June 2000.

[18] P. Tarr, L. Bergmans, M. Griss, and H. Ossher. Workshop on Advanced Separation of Concerns, OOPSLA 2000. Minneapolis, USA, October 2000. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/.

[19] P. Tarr, M. D'Hondt, L. Bergmans, and C.V. Lopes. Workshop on Aspects and Dimensions of Concerns: Requirements on, and Challenge Problems for, Advanced Separation of Concerns, ECOOP 2000. In J. Malenfant, S. Moisan, and A. Moreira, editors, *ECOOP 2000 Workshop Reader*, LNCS 1964, page 203 ff., Sophia Antipolis and Cannes, France, June 2000.

[20] A. Zeller and G. Snelting. Unified Versioning through Feature Logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398 – 441, 1997.