

A Version Model for Aspect Dependency Management

Elke Pulvermüller¹, Andreas Speck², and James O. Coplien³

¹ Institut für Programmstrukturen und Datenorganisation,
Universität Karlsruhe,
D-76128 Karlsruhe, Germany
pulvermueller@acm.org

² Intershop Research Jena
D-07740 Jena, Germany
andreas.speck@intershop.com

³ Bell Laboratories Lucent,
Naperville IL, USA
cope@research.bell-labs.com

Abstract. With Aspect-Oriented Programming (AOP) a new type of system units is introduced (aspects). One observed characteristic of AOP is that it results in a large number of additional (coarse-grained to fine-grained) system units (aspects) ready to be composed to the final application. With this growing number of system units the dependencies between them become vast and tangling. This results in the necessity of an improved management of the dependencies between these system units. Our paper investigates this problem, proposes a more general model (version model) to capture different facettes 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 (e.g. object-oriented) software engineering allowing an improved realization of the “Separation of Concerns” principle [26]. Besides classes and objects, aspects are an additional type of system units. They serve to localize any cross-cutting code, i.e. code which cannot be encapsulated within one class but which is tangled over several classes [20]. Therefore, an aspect is a specific type of concern, namely a cross-cutting concern [19].

Beyond the identification and definition of aspects a further problem arises: How can an aspect configuration be verified and how can the correctness of their mutual dependencies and interactions be proved? The more aspects 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 [32], [31] and [28].

Examples of dependencies in practice are:

- Telecommunication domain:
A telephone switching station has to provide many different services. Some of them are cross-cutting and may be captured in aspects. Inserting aspects into the running system to extend the functionality may lead to interactions between the newly inserted units and the system units already part of the system. Research exists dealing with that problem (but still without employing supportive AOP concepts) [12].
- Aspects in distributed systems [27]:
In CORBA systems the communication code is strongly interwoven with the application code. 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 [30]:
In the growing market of small embedded systems software is often reused when the same control hardware is modified in order to perform similar services. 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. Besides a few adaptations of the software statements the software for all these devices may be reused. 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 data-flow statements missing (especially with respect to security and error handling code). Similar problems may be found when the systems have to be adapted to different often very similar field-bus protocols.
- In the AOP workshop at Rennes [19] L. Seiturier presented a Virtual Virtual Machine (VVM) [6]. A VVM is a micro-VM loading VMlets. A VMlet is similar to an Applet but describes an execution environment instead of an application. This technology provides an open virtual machine, a (micro) runtime, where each domain expert may build his/her specific execution environment adding or modifying functionality (by means of VMlets). Some of the VMlets may be aspects. The VVM, therefore, weaves aspects into a base system.
However, no coordinator or similar technology is implemented to avoid clashes between the separately loaded VMlets. If a VMlet is dependent (e.g. via initialization) from another one an error occurs.

In the remainder of the paper we introduce a version model and show how it captures and extends the idea of AOP in a more general way and, moreover, how it provides a means to preserve consistency and correctness. The approach presented is not limited to AOP but may be employed for any kind of feature composition. Section 3 examines the consistency and correctness conditions in more detail. Some related work may be found in section 4.

2 Version Model

Aspects may cross-cut object systems [20] on various levels of granularity: For instance, 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.

Our approach for the aspect dependency problem is based on a broader viewpoint onto the different granularity levels of aspects: 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.

The versions we discuss in our approach may be stored in versioning systems as elaborated e.g. in [13], [23] or [5]. Note that in contrast to that work, we do not focus on the administration and storage of different versions of software systems. In our approach versions are issues of system construction with aspects. However such versioning systems may be applied to manage the versions of our approach. The described version model and existing systems or research in the area of versioning systems, respectively, may complement one another.

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.

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:

$$\text{Version } V_i^0 = 1 \wedge \text{Cond}_i^0$$

where Cond_i^0 represents the condition ¹ that has to be true for one version V_i^0 on level 0. $V_i^0 \in V^0$ where

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

is the set of all versions on level 0.

¹ Several conditions may be unified in one condition.

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

$$\begin{aligned} &Cond_i^{(n+1)} \text{ is true,} \\ &1 \leq l \leq m \leq |V^n|, \\ &V_j^n \in V^n \end{aligned}$$

□

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 (cf. section 3).

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

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

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). In fact, the border between these two poles is not fixed but is viewpoint respectively design dependent.

Some of the atomic conditions (in the first case), for instance, have the following appearance: $C1 =$ “Statement 1 (code line 1) has to be in the aspect” = $S1$, $C4 =$ “Statement 4 has to be in the aspect” = $S4$. 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”, $A2 =$ “Aspect with ID $A2$ has to be part of the system”. There are different possibilities to uniquely identify an aspect (e.g. by name or by unique identifier).

Figure 1 shows an example for the second problem discussed in section 1. We demonstrate the usage of versions to model the construction:

– **Example for modeling individual aspects (fine-grained):**

Based on a file containing statements (lines of code) as depicted in figure 1 a version reflecting an individual aspect may be constructed as follows.

$$Version V_1^1 = 1 \wedge C_1^1$$

with $C_1^1 = C1 \wedge C2 \wedge \neg C7$ and the atomic conditions $Cn =$ “Statement n has to be part of the aspect” = Sn , $n \in \{1, 2\}$ and $C7 =$ “System is built for France” (additional condition).

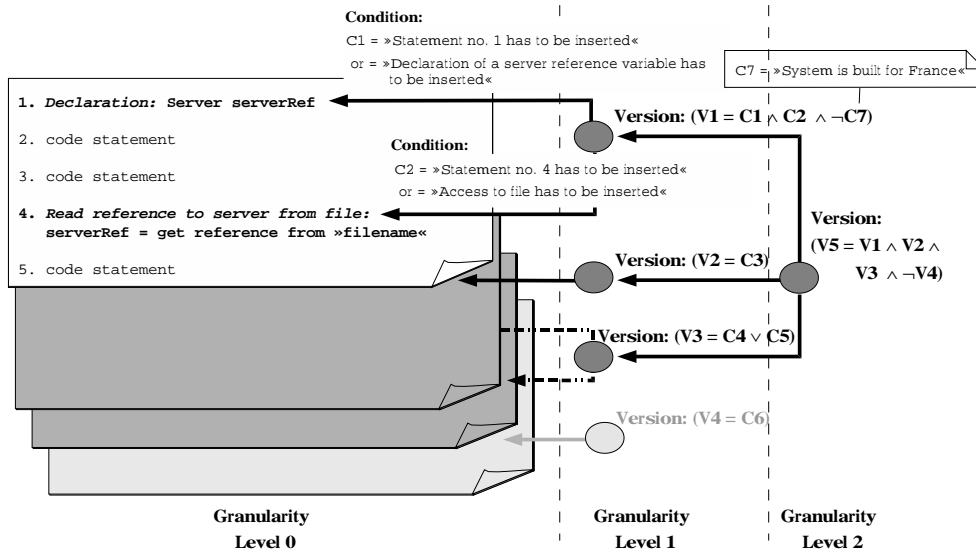


Fig. 1. Versions in a Distributed System

Note that we start with granularity 1 in this case. Granularity level 0 reflects single statements or their existence respectively.

Besides the construction of aspects, the version model also serves to express the proper combination of aspects which is a logical and consistent continuation of the first case but on a higher granularity (cf. the following example).

– **Example for modeling a set of aspects forming a valid configuration (coarse-grained):**

Figure 1 contains a version (V_5 respectively V_5^2) for a distributed system built from individual aspects. 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 otherwise these versions are not valid. E.g. V_1^1 may represent the file-based reference of the server (expressed by C_1^1 and C_2^1) from non-French systems (represented by $\neg C_7^1$).

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 language and technique realizing the concrete aspects and software system (e.g. AspectJ [1], HyperJ [3], meta-programming approaches or transformation systems [15], [24], [2], [10], respectively). 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 (then the boolean expression of the version is false).

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 [21]. Besides the reuse of the aspects and additional system issues it becomes possible to use existing condition information to decide whether the reuse is semantically reasonable in a certain context.

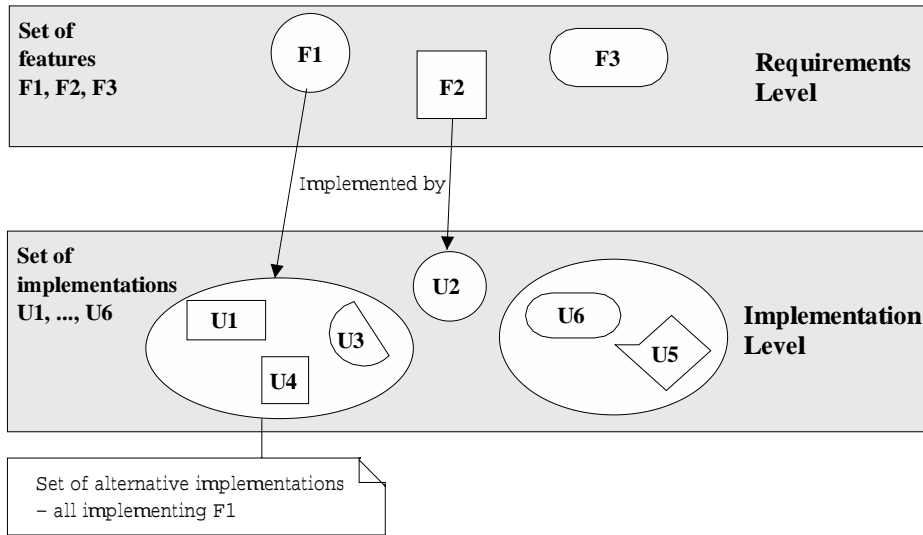


Fig. 2. Relationship between Requirements and Implementation Level

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

Formally the relationship can be expressed as follows:
 $U1, U2 \in Set1$; $F1, F2 \in Set2$ where $Set1$ is the set of implementation units and $Set2$ is the set of features on the requirements level. Let us assume that $U1$ implements (besides others) feature $F1$ and $U2$ implements $F2$ then the following property holds:

$$valid(U1, U2) \Rightarrow valid(F1, F2)$$

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

with function

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

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 aspects 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 piece-meal growth manner [14].

Domain engineering [16] 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 [16], 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

Complementary work may be found in [21]. The Hoare triples (or pre- and post-conditions as its realization mechanisms respectively) are employed to guarantee valid aspect combinations during the combination process. While [21] concentrates on the checking mechanism and assumes given aspect combination rules without focusing on the rules themselves the approach in this paper provides a version model to structure the rules or conditions respectively. This is a step towards improved mathematical support dealing with a complex set of rules and also towards their (partial) automatic processing.

Related work may be found in all aspect-oriented and related approaches like subject-oriented programming [18], adaptive programming [22], adaptive plug & play components [25], composition filters [7] or also transformation systems [10].

Common to all these approaches is the goal to divide a system into smaller units in a more natural way aiming at reducing the gap between code and reality and managing the inherent complexity.

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 instruction). 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 in a natural way.

With respect to composition validation further work may be found in [9]. In the GenVoca model composition is described with type equations. A design rule checking

mechanism detects illegal combinations. GenVoca is a powerful layered model and thus mainly layered composition is considered.

Prior work at Bell Laboratories and in [29] about versioning has influenced the proposed version model approach.

Generative programming [16] is another related field which may augment the version model. The version notation may serve as an input for generators. Especially domain engineering as one part of the generative programming approach is, vice-versa, a means to detect, collect and describe semantic conditions. Feature models may be an important technique in this context.

Requirements engineering methods in general bear a potential to detect and explore semantic knowledge which is needed in the version model and captured in the conditions.

Also, AI technologies like expert systems may be used. [17] describes a way to use expert systems to support reuse of object-oriented frameworks by means of explicitly encoded design knowledge and user interaction. Aspect composition conditions are one type of such semantic design knowledge.

An alternative approach using logic to describe aspect dependencies may be found in [11].

University of Twente currently conducts research on fuzzy quantification of domain knowledge which is a promising complement to the presented work [8].

5 Summary and Conclusion

The version model proposed in this paper allows both, to describe the internal structure of an aspect and to define valid clusters of aspects. 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 from 0 to 1. Semantic conditions may have a value in a range of discrete or analogue values. Currently, further research is going on in this area together with University of Twente in the context of the European project EASYCOMP [4].

References

1. *AspectJ - Aspectj-Oriented Programming (AOP) for Java*. <http://www.aspectj.org>, 2001.
2. *COMPOST Homepage*. <http://i44w3.info.uni-karlsruhe.de/~compost/>, 2001.
3. *HyperJ: another alphaWorks technology*. <http://www.alphaworks.ibm/tech/hyperj>, 2001.
4. *IST Project 1999-14191 EASYCOMP*. <http://www.easycomp.org>, 2001.
5. *RCE, VRCE, BDE; RCE: the Revision Control Engine*. <http://www.ipd.ira.uka.de/~RCE/>, 2001.
6. *Virtual Virtual Machines*. <http://www-sor.inria.fr/projects/vvm/>, 2001.
7. M. Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4), December 1996.
8. Tekinerdogan B. Design and Experimentation of a Fuzzy Logic Controller for Evaluating Domain Knowledge. In *Proceedings of Second International Workshop on Softcomputing Applied to Software Engineering (SCASE)*, 2001.

9. D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, pages 67 – 82, 1997.
10. I. Baxter. Design Maintenance Systems. *Communications of the ACM*, 35(4):73 – 89, April 1992.
11. J. Bricau. Declarative Composable Aspects. In *Proceedings of Workshop on Advanced Separation of Concerns, OOPSLA*, 2000.
12. M. Calder and E. Magill. Proceedings of Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems. IOS Press, 2000.
13. R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232 – 282, 1998.
14. 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.
15. 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.
16. K. Czarnecki and U.W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.
17. W.D. De Meuter, M. D'Hondt, S. Goderis, and T. D'Hondt. Reasoning with Design Knowledge for Interactively Supporting Framework Reuse. In *SCASE*. <http://progwww.vub.ac.be/Research/ResearchPublicationsDetail2.asp?paperID=81>, February 2001.
18. Osher 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.
19. IRISA/IFSIC. Workshop on Aspect Oriented Programming, co-located with OCM, Objets, Composants et Modeles. Rennes, France, May 2001. <http://www.irisa.fr/coo/OCM2001/programAOP.htm>.
20. 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.
21. 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.
22. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
23. E. Lippe and G. Florijn. Implementation Techniques for Integral Version Management. In *Proceedings of ECOOP'91, European Conference on Object-Oriented Programming*, LNCS 512. Springer, 1991.
24. 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.
25. M. Mezini and K.J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *ACM SIGPLAN notices*, volume 33, October 1998.
26. D.L. Parnas. On The Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972.

27. 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.
28. 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. <http://i44w3.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/>. To be held.
29. M.J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364 – 370, December 1975.
30. 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.
31. 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/>.
32. 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.