In *Proc. of Generative and Component-based Software-Engineering Second International Symposium GCSE 2000*,
Springer, LNCS 2177, pages 57-69, Erfurt, Germany, 2000

Aspect Composition applying the Design by Contract Principle

Herbert Klaeren[1], Elke Pulvermüller[1], Awais Rashid[3], and Andreas Speck[1]

[1] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
D-72076 Tübingen, Germany
{klaeren, speck}@informatik.uni-tuebingen.de
[2] Institut für Programmstrukturen und Datenorganisation,
Universität Karlsruhe,
D-76128 Karlsruhe, Germany
pulvermueller@acm.org
[3] Computing Department,
Lancaster University,
Lancaster LA1 4YR, UK,
marash@comp.lancs.ac.uk

Abstract. The composition of software units has been one of the main research topics in computer science. This paper addresses the composition validation problem evolving in this context. It focuses on the composition for a certain kind of units called aspects. Aspects are a new concept which is introduced by aspect-oriented programming aiming at a better separation of concerns. Cross-cutting code is captured and localised in these aspects. Some of the cross-cutting features which are expressed in aspects cannot be woven with other features into the same application since two features could be mutually exclusive. With a growing number of aspects, manual control of these dependencies becomes error-prone or even impossible. We show how assertions can be useful in this respect to support the software developer.

1 Introduction

Composing systems from individual units has been one of the main research goals in the software engineering discipline since its beginning in the 1960s. The first approaches concentrating on modules were followed by decomposition into objects and / or components [23, 4, 30, 29]. All of these approaches aim at managing complexity by dividing a system into smaller pieces.

Once a systems analyst completes the creative process of dividing the system into manageable parts, phases follow where the units are realised and synthesised to form the final system. The units themselves are built from scratch or reused if they already exist provided they can be used in the chosen technical environment.

It has become popular to build systems not only for one purpose but to allow variability to a certain extent. This is due to the goal to save development and maintenance effort and increase software quality by reusing a system multiple times even in slightly different contexts. This development is reflected in the intensified research in the field of product line architectures (PLA) [27]. In a concrete context the necessary and available units are reused and the points of variability are adapted to the particular needs [10, 8]. This results in a set of valid configurations for one kind of system. The more configurations exist the more flexible is the system concerning reuse and adaptation. However, an increase in variability and flexibility poses a greater challenge in validating the configuration of chosen units. The search for a valid configuration is error-prone if manually practised. Therefore, there is a need for computer support. Such computer support is also realisable since rules can be used to specify valid configurations in advance.

Aspect-oriented programming (AOP) introduces a new concept called aspects [17]. Besides classes, they form an additional type of system unit. Aspects serve to localise any cross-cutting code, i.e. code which cannot be encapsulated within one class but which is tangled over many classes. The existence of this new type of system unit affects the composition validation. Although some composition validation mechanisms already exist [3, 7], aspects have not been explicitly considered. In this paper we introduce a composition validation mechanism which in particular concentrates on this new kind of system units.

In our approach we use the technique of assertions [21]. Assertions are a widely accepted programming technique for complex systems to improve their correctness. Taking the validity of a configuration as a property, assertions can be used to ensure that the chosen aspects and classes fit together. In the following, we provide further background information about aspect-oriented programming and assertions which is necessary to understand the remainder. A motivating example demonstrates the problem we address. Sections 3 to 5 describe our different aspect composition validation approaches based on assertions. Section 6 concludes the paper and identifies directions for future work.

2 Background and Motivation

This paper is based on two techniques, i.e. aspect-oriented programming and assertions. Here, assertions are used as a means to validate aspect composition. In the following, the term "aspect composition" is used for the insertion of a set of aspects into one system / class. This set is also called (aspect) configuration.

Aspect-oriented programming (AOP) [17] introduces a new concept, called aspects, in order to improve separation of concerns. It aims at easing program development by providing further support for modularisation at both design and implementation level. Objects are designed and coded separately from code that cross-cuts the objects. The latter is code which implements a non-functional feature that cannot be localised by means of object-oriented programming. Code for debugging purposes or for object synchronisation, for instance, is tangled over

all classes whose instances have to be debugged or synchronised, respectively. Although patterns [12] can help to deal with such cross-cutting code by providing a guideline for a good structure, they are not available or suitable for all cases and mostly provide only partial solutions to the code tangling problem. With AOP, such cross-cutting code is encapsulated into separate constructs or system units, i.e. the aspects. The links between objects and aspects are expressed by explicit or implicit *join points*. An *aspect weaver* is responsible for merging the objects and the aspects. This can be done statically as a phase at compile-time or dynamically at run-time [17, 15]. For our research, we have used AspectJ0.4beta7 from Xerox PARC [32, 18] for all AOP implementations together with Java JDK 2.

Although AOP allows to achieve a better separation of concerns, the application of aspect-oriented programming bears new challenges. This paper addresses one of them: aspect composition validation.

Let us assume we have an application and a set of aspects which can be woven with the objects of the application. This set may contain aspects which are redundant or even exclusive with respect to other aspects in the set. Alternatively, weaving one aspect may require another aspect to be woven. For example, if there are two different aspects `A1` and `A2` containing debugging functionality, they may be mutually exclusive. `A1` inserts code into the application which realises all tracing. Each function call is recorded in a window with graphical support for user control. As opposed to graphical debug support, `A2` realises the same tracing functionality as ASCII output. Since both aspects implement the same feature (although in different ways) it wouldn't be reasonable to weave both aspects. Depending on the context, the environment or the requirements (e.g. towards efficiency), the one or the other is more suitable.

In [25, 28] an example implementation can be found which also reveals this aspect composition problem.

At this point, assertions [21, 26] can be used to achieve correctness. The basic ideas behind assertions originate in the theory of formal program validation pioneered by R.W. Floyd [11] C.A.R. Hoare [14] and E.W. Djikstra [9]. The Hoare triples provide a mathematical notation to express correctness formulae. Such a correctness formula is an expression of the form:

$$\{P\}A\{Q\} \tag{1}$$

This means that *"any execution of A, starting in a state where P holds, will terminate in a state where Q holds"* [21]. In terms of software, A denotes an operation or software element whereas P and Q define the assertions, or pre- and postcondition, respectively. By means of assertions it becomes possible to state precisely the formal agreement between caller / client and callee / supplier, i.e. both, what is expected from and what is guaranteed to the other side. This reflects the "Design by Contract" principle [20].

For our purposes, the validity of a set of aspects which are woven in A reflects an assertion.

3 Aspect Composition Validation

Since correctness is always relative [1] [21] we assume to have a certain specification defining (among other things) which features have to be realised in the application. Therefore, the decision which feature (i.e. here which aspect) should be inserted (i.e. woven) is not part of our solution. Moreover, the knowledge whether a set of aspects is valid (in this paper also called valid configuration) or not, has to be extracted during the analysis phase and captured in the specification (e.g. using finite state automata as outlined in [24]) before assertions can be useful to prove this property.

According to the definition of assertions, we use them to verify that a class or an application contains a suitable set of aspects which forms a valid configuration (with respect to what is written in the specification). With these assertions, the specification is expressed and included in the implementation to assess the correctness of aspect composition. In this paper, we concentrate on this issue of software correctness and omit all considerations about other properties which can be asserted as already known [13].

Expressing the part of the specification concerning the aspect composition within the code can serve various purposes. These range (similar to other kinds of assertions) from treating assertions purely as comments with no effect at run-time to checking the assertions during execution.

Sections 3.1 and 3.2 describe different ways to realise assertions checking for a valid aspect configuration.

3.1 Precondition, Postcondition or Invariant

B. Meyer defines an assertion as *"an expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution"* [21]. In our case, the entities of the software involved are the classes or methods whereas the stated property is the validity of the aspect configuration injected into these classes or methods through the aspect weaver.

Three possibilities to assert this property can be identified: preconditions, postconditions or class invariants [21].

First, this property can be asserted as a precondition. The class or method starts its work assuming that a valid set of aspects is woven and active. The precondition checks whether this assumption is true, i.e. whether the contract is fulfilled by the caller and the property to have a valid aspect configuration in the callee holds (cf. figure 1 on the left). This may be reasonable if the caller changes the aspect configuration which is active in the callee (here we assume that the aspects themselves are not changed, i.e. there is no aspect evolution). Applying the design by contract principle, the caller ensures that the changed aspect configuration is valid for the callee. In the precondition the callee checks whether the caller kept the contract.

[1] This is also expressed by P in equation (1).

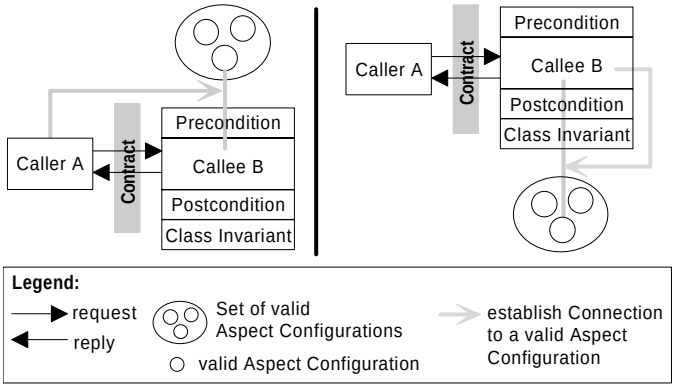

Fig. 1. Different Possibilities to assert a valid Aspect Configuration.

Although this is possible, we propose [2] to assert the validity of an active aspect configuration in postconditions or class invariants. While pre- and postconditions describe the properties of individual routines, class invariants allow to express global properties of the instances of a class which must be preserved by all routines. An invariant for a class is a set of assertions (i.e. invariant clauses) that every instance of this class will satisfy at all times when the state is observable. The crucial point is that with both, postconditions and invariants, it is a bug in the callee if the assertion is violated [21]. We believe that in most cases it is more reasonable that the decision about which aspect should be active in a class C at which times is in the responsibility of C. Thus, in our understanding, the callee should decide by itself which aspect instances should be connected to this callee. Consequently, as opposed to a precondition realisation, the callee is also responsible to ensure and to check that the woven aspects which are active in the callee form a valid configuration. Obviously the same argumentation applies symmetrically to the caller which can also be a callee.

Thus, the remainder of the paper concentrates on postconditions and class invariants. It goes without saying that the principles shown are also applicable to preconditions.

3.2 Static or Dynamic

There are two ways to assess software correctness. The property of a class or method to result in a valid aspect configuration can be ensured at run-time (dynamically) or alternatively at (or before) weave-time (statically).

A similar distinction can be identified if the times of aspect configuration changes are considered. Either it is possible to add or remove aspect instances to

[2] It should be avoided to assert a property multiple times due to the disadvantages of redundant checks and defensive programming [21].

or from class instances only statically or even dynamically (cf. table 1; + indicates that this combination is possible whereas ++ expresses that this combination is preferable provided there is a choice).

		Change of Aspect Configuration	
		static	dynamic
Aspect Composition Validation	dynamic	+	+
	static	++	usually not possible

Table 1. Static and dynamic Change and Validation.

In the following we show the outlined differences between *static* and *dynamic* at some AspectJ0.4beta7 / Java JDK 2 code extracts (cf. figure 2).

In AspectJ0.4beta7, each aspect has (like classes) its name following the keyword `aspect` and contains the advised methods with their names and the class names referring to these methods. Besides method advising [3], it is also possible to introduce whole methods.

Dynamic or static Change of Aspect Configuration

At first, the change of the set of aspects which play an active role in a class instance can be done at weave-time without later changes during execution time. This situation is depicted in figure 2 on the left. These static connections between an aspect and all instances of a class are expressed with the keyword `static` in AspectJ. Once the weaver injected these static aspects into class `StaticExample` at weave-time, the functionality within these aspects will be active in all instances of `StaticExample` during the whole execution time. The woven aspects augment the class code of `StaticExample` which impacts all its instances. Moreover, it is not possible either to add further static aspects nor to remove statically woven aspects from one or all instances during run-time.

As opposed to statically woven aspects, it is also possible to create new aspect instances and connect them to objects during run-time. In figure 2, this is shown on the right. The code within the light-grey lined rectangle in the `DynamicExample` class [4] establishes the connection between aspects and class instance during run-time. The commands used (e.g. `addObject(...)`) are provided by AspectJ0.4beta7. If other AOP environments are used, similar language constructs are necessary. Alternatively, dynamic weaving capabilities can be used if available [15].

[3] By using the keyword `before` the aspect weaver injects this advised code at the beginning of the method. The keyword `after` augments the method at the end.

[4] For demonstration purposes we have chosen to present an aspect-directional design [16], i.e. the class knows about the aspect but not vice-versa.

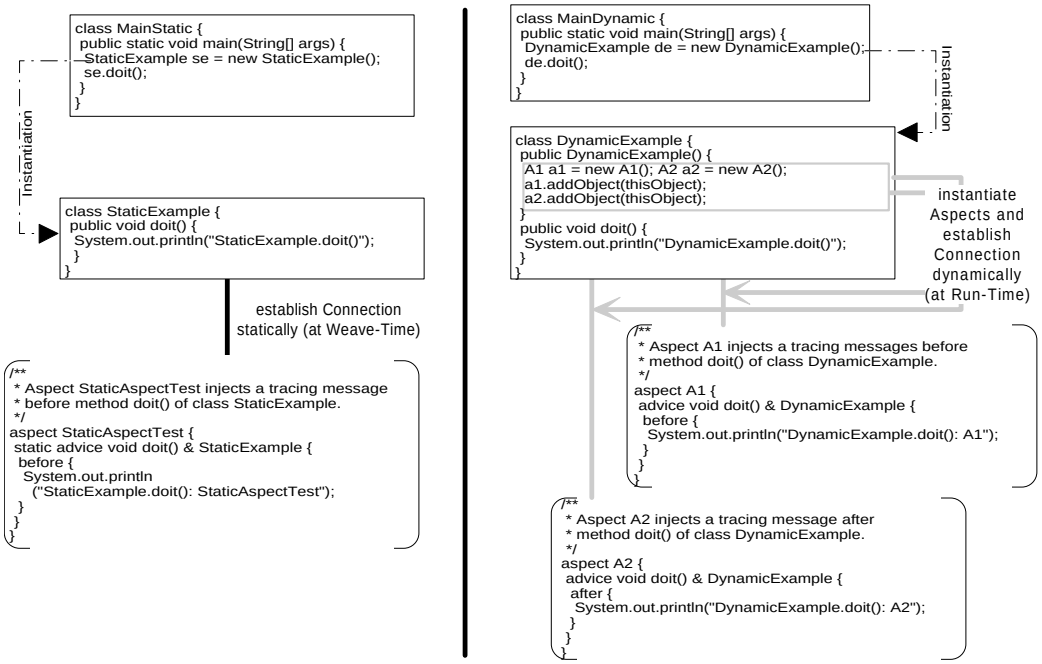

Fig. 2. Example with statically or dynamically connected Aspects.

Dynamic or static Aspect Composition Validation

Assertions express correctness conditions (here: validity of the aspect configuration). Assertion rules check whether such a condition is violated. These rules can be executed during run-time. If possible, a violation check can also be done statically. This requires to have all the necessary information at compile-time or weave-time which is true for statically woven aspects. During execution the static aspect configuration does not change. Verifying and removing assertions statically has the advantage that the overhead of the test during execution is avoided [13]. Thus, although statically connected aspects can be verified at run-time (in table 1 depicted with $+$), assertions execution at compile-time or weave-time is to prefer (this is expressed by $++$ in table 1). Static assertion violation checking is described in section 5.

Dynamically changing aspect configurations (as depicted on the right in figure 2) can hardly be checked at compile-time or weave-time. Which aspect instance is created and connected to which class instance depends on the dynamic control flow and user input. In this case, the dynamically executed assertions are usually unavoidable. The technique for dynamically asserting aspect configurations is described in the next section.

4 Asserting dynamically changing Aspect Configurations

In this chapter, we concentrate on all aspect composition validation which cannot be asserted statically as outlined above. Dynamically created and connected aspect instances have to be checked during program execution. These assertions can be added into the corresponding class. Since aspects can be used to separate assertion functionality from problem domain-related code, it is also possible and reasonable to extract the aspect configuration assertion code from the class and localise it into a separate assertion aspect. Thus, if a class wants to test whether the postcondition (i.e. the property "a valid configuration of active aspects is connected") holds, an assertion aspect has to be woven. As the other aspects, this aspect can be instantiated and connected dynamically or statically. The implementation of such a static assertion aspect for `DynamicExample` of figure 2 is outlined in figure 3.

```
/*
 * Aspect AspectAssert injects assertion functionality after the constructor of
 * DynamicExample. Thus it is asserted  that the changed aspect configuration is valid.
 */
aspect AspectAssertion {
 introduction DynamicExample {
  // -- The facts or knowledge base and the rules --
  private static boolean not(String a) {
   if (a.equals("A3")) return true; // not A3
   if (a.equals("A4")) return true; // not A4
   return false;
   }

  private static boolean xor(String a, String b) { ... }

  // -- Check if any facts and composition rules are violated --
  private static boolean check(java.awt.List aspect_names) { ... }

  public boolean assert_composition() {
   // Obtain aspect references of aspects which are active:
   java.util.Vector v = thisObject.getAspects();
   // Derive "aspect_names" list with the aspect names from v
      ...
   boolean b = check(aspect_names);
    return (b);
   }
  }

  // -- Postcondition is injected by augmenting the constructor --
  static advice void new(..) & DynamicExample {
   after {
   if (assert_composition()) throw new PostConditionViolation();
   } catch (Exception e) { System.out.println(e.toString()); }
   }
}
```

Knowledge Base and Rules defining valid Aspect Configuration

Check active Configuration against Knowledge Base and Rules

Assertion

Fig. 3. Assertion Aspect for dynamic Testing.

Note that with the introduction of these new assertion aspects, these aspects themselves may be ensured by higher-level assertion aspects. This situation can be compared to the abstraction hierarchy in object-orientation ("instance", "class" and "meta-class"). The crucial question is how many abstraction levels are reasonable. These considerations are beyond the scope of this paper [31].

According to figure 3, the knowledge base and rules part expresses the knowledge contained in the specification. It defines all valid aspect configurations. The actual configuration is determined with the AspectJ command

`thisObject.getAspects()` during run-time which returns all active aspect instances connected to the class instance. Such a possibility to determine all woven and active aspect instances at run-time is crucial to their dynamic assertion. This determined set of active aspect instances is then checked against the knowledge base and rules. If this check proves that the postcondition is true, an invalid aspect configuration is detected and an exception is raised (cf. figure 3).

An interesting part is the one expressing the knowledge base and the rules of a valid aspect configuration. Some basic types of rules can be identified which allow to express the relevant dependencies between the aspects and between the aspects and a certain class instance (with respect to the specification). For our various example implementations the following rules proved to be sufficient:

- `not A1`: A not-clause expresses that the aspect with name `A1` is not allowed to be woven into the class instance.
- `requires A1`: Note that this term differs from *require* clauses sometimes used to express assertions (e.g. in the programming language Eiffel [19]). Here, it is meant that weaving aspect `A1` into the class instance is mandatory. This rule may also have multiple arguments (e.g. `requires A1 A2 A3`) which indicates that at least one of these aspects is mandatory (in the example either `A1` or `A2` or `A3` is mandatory).
- `xor A1 A2`: This expression indicates that either `A1` or `A2` may be woven into a class instance but never both of them.
- `and A1 A2`: An `and`-clause expresses that `A1` and `A2` have to be woven together into the same class instance.

In figure 3 the methods `not` and `xor` outline a possible implementation of two types of rules in AspectJ0.4beta7 / Java JDK 2. The `check(...)` method describes the application of these rules to a set of aspects.

Although these dependency rules are expressed in AspectJ0.4beta7 / Java syntax in the presented implementation a multi-paradigm approach [6, 5] would be suitable here. With a logic programming language (e.g. Prolog) the implementation of the knowledge or dependency rules contained in the specification would lead to improved understandability. Generally speaking, a domain-specific language [7] based on the predicate calculus would improve the implementation. Alternatively, a domain-specific language based on finite state automata can be used to express these dependencies as described in [24].

An assertion aspect for a certain software system may be also generated from a file containing the knowledge base and rules [28].

5 Asserting static Aspect Configurations

While the validation of the dynamic aspects results in performance penalties, these can be avoided for static aspects since the configuration can be assured before the execution (cf. table 1). It is a common technique to remove assertions at compile-time provided they can be checked in a static analysis [13].

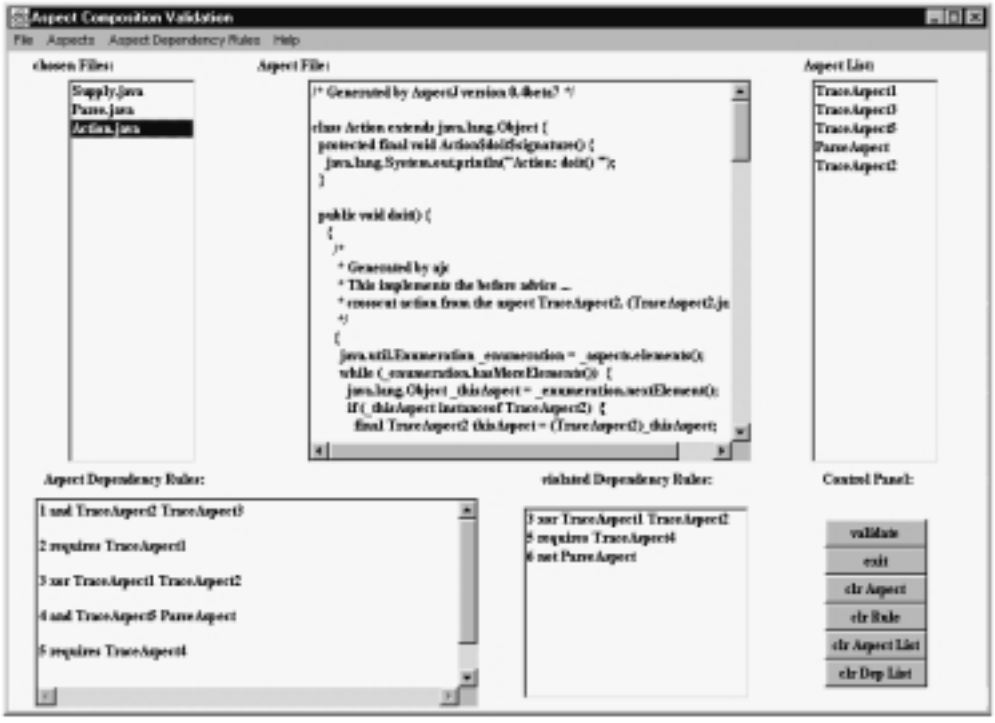

Fig. 4. *Aspect Composition Validation* Tool.

The principle of such static validation procedure is as follows: Assuming we have a class implementation including static assertion clauses to assure a valid aspect configuration. Then, the compiler, weaver or any tool operating before execution time can examine this class implementation. The statically known information about what has to be asserted can be extracted from the code and checked before run-time. The dynamic tests during execution can be avoided since the already statically checked assertions can be eliminated.

Based on this observation, we built the "*Aspect Composition Validation*" tool (cf. figure 4) in order to automate the static verification. The tool realises a slightly different verification procedure compared to the principle described above. We chose not to inject assertion code into the classes. This eliminates the need to remove this code before execution. The developer decides by menu which classes the tool has to assert. The assertion itself (i.e. the property that is to be asserted) is obvious: the aspects connected to the class have to be checked according to aspect dependency rules derived from the specification.

The tool can be used to ensure the property of a valid static aspect configuration within a specific class. Such a class-wide validation corresponds to the concept of class invariants described in [21]. Moreover, the tool is also able to

verify the aspect configuration of a set of classes. Such a set may be a package, a component or a subsystem. Therefore, this is an extension of B. Meyer's understanding of assertions (which is limited to pre-, postconditions and class invariants) to more than a single class, i.e. to more coarse-grained building blocks of the system. In the tool, the single file or the set of files containing the class(es) to be validated are listed in the *chosen Files* list with their aspects displayed in the field *Aspect List* as shown in figure 4. Additionally, the highlighted file in the file list is presented in the *Aspect File* text area.

The tool verifies the aspect configuration (consisting of all the aspects which are included in the listed files on the left) according to the aspect dependency rules (initiated by pressing the *validate* button). Violated aspect dependency rules are displayed in the list named *violated Dependency Rules*. The aspect dependency rules expressing the invariants of aspect combination are the same as those described in section 4 (`not`, `requires`, `and` and `xor`). The software developer can insert or change these aspect rules directly in the *Aspect Rules* text area. Alternatively, they can be read from a user-defined file containing these rules written in the domain-specific language.

The *Aspect Composition Validation* tool extracts the static aspects by parsing the files containing the already woven source. AspectJ0.4beta7 marks all woven code sections with comments. Alternatively, the woven aspects could be derived from specific documentation files (with extension `.corr`) provided by AspectJ0.4beta7. This file documents which aspect is woven in which class or method. The woven aspect set could also be obtained from the original source files directly.

6 Conclusion

Although aspect-oriented programming can improve software due to a better separation of concerns, the software developer is faced with new challenges. One of them is the aspect composition validation which is not yet examined sufficiently in AOP and therefore is addressed in this paper.

Both the dependencies between the aspects woven with a class instance and the dependencies between these woven aspects and the class instance itself have to be identified in a specification. On this basis we demonstrate how to use assertions to ensure the correctness of these dependencies with respect to the specification. Faulty aspect configurations (e.g. if the set of woven aspects embraces aspects which are contradictory) can be detected using assertions similar to other bugs. Dynamically changing aspect configurations are checked at runtime. Due to the performance penalty in case of dynamic tests, we also presented a static analysis which is preferable in case of static aspect configurations. Although the feasibility of the principles and concepts in this paper is shown with AspectJ0.4beta7 and Java JDK 2 implementations the approach is independent of the concrete technology. For instance, using composition filters [2], [1] to achieve a better separation of concerns also leads to the problem that both the

order of the filters and the filters of the different objects within one application have to fit semantically. Assertions can be used similarly in this case.

Our further research will concentrate on transferring these concepts to component or object composition validation with respect to already existing support in this domain [3]. There is a growing need for computer support in the field of composing systems of reusable parts which are stored in repositories. As with aspects, there might be multifarious dependencies between these parts. Another field is the graphical representation of the dependencies and of all violated dependency rules within a certain application. Besides this graphical feature, a tool environment can include further support for the software developer. The description of a valid configuration, i.e. of the knowledge base and the rules according to the specification, should be separated. Further investigation in the sense of domain engineering is necessary there. Moreover, a partial automatic generation of assertion aspects from such separated descriptions is possible and will be another research activity in our future work.

References

1. M. Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4), December 1996.
2. M. Aksit and B. Tekinerdogan. Aspect-Oriented Programming Using Composition Filters. In *ECOOP 1998 Workshop Reader*, page 435. Springer-Verlag, 1998.
3. D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, pages 67 – 82, 1997.
4. G. Booch. *Object-Oriented Analysis and Design.* Benjamin/Cummings, Redwood City, CA, second edition, 1994.
5. J.O. Coplien. Multi-Paradigm Design. In A. Speck and E. Pulvermüller, editors, *Collection of Abstracts of the GCSE99 YRW*, `http://www-pu.informatik.uni-tuebingen.de/users/speck/GCSE99_Young_Research/abstracts/Jim_Coplien_gcseYR99.html`, September 1999.
6. J.O. Coplien. *Multi-Paradigm Design for C++.* Addison-Wesley, 1999.
7. K. Czarnecki. *Generative Programming, Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technical University of Ilmenau, Ilmenau, Germany, 1999.
8. K. Czarnecki and U.W. Eisenecker. Synthesizing Objects. In *Proceedings of ECOOP'99*, Lecture Notes in Computer Science LNCS 1628, pages 18 – 42. Springer-Verlag, June 1999.
9. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
10. U.W. Eisenecker. Generative Programming GP with C++. In H.-P. Mössenböck, editor, *Proceedings of Joint Modular Programming Languages Conference*, LNCS 1204. Springer-Verlag, 1997.
11. R.W. Floyd. Assigning Meanings to Programs. In J.T. Schwartz, editor, *Proc. Am. Math. Soc. Symp. in Applied Math.*, volume 19, pages 19 – 31, Providence, R.i., 1967. American Mathematical Society.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstractions and Reuse of Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.
13. J. Gough and H. Klaeren. Executable Assertions and Separate Compilation. In H.-P. Mössenböck, editor, *Proceedings Joint Modular Languages Conference*, LNCS 1204, pages 41 – 52. Springer-Verlag, 1997.

14. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576 – 583, October 1969.
15. P. Kenens, S. Michiels, F. Matthijs, B. Robben, E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. An AOP Case with Static and Dynamic Aspects. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98*, 1998.
16. M. A. Kersten and G. C. Murphy. Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. OOPSLA, 1999.
17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science LNCS 1241*, ECOOP. Springer-Verlag, June 1997.
18. C. V. Lopes and G. Kiczales. Recent Developments in AspectJ. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98*, 1998.
19. B. Meyer. *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, 1991.
20. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40 – 51, October 1992.
21. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1997.
22. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998.
23. D. L. Parnas. On The Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972.
24. L. Pazzi. Explicit Aspect Composition by Part-Whole Statecharts. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, 1999.
25. E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in Distributed Environments. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering GCSE'99*, Erfurt, Germany, September 1999.
26. D.S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transaction on Software Engineering*, 21(1):19 –31, January 1995.
27. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Lecture Notes in Computer Science LNCS 1445*, pages 550 – 570, 1998.
28. A. Speck, E. Pulvermüller, and M. Mezini. Reusability of Concerns. In C. V. Lopes, L. Bergmans, M. DHondt, and P. Tarr, editors, *Proceedings of the Aspects and Dimensions of Concerns Workshop, ECOOP2000*, Sophia Antipolis, France, June 2000.
29. C. Szyperski. *Component Software*. Addison-Wesley, ACM-Press, New York, 1997.
30. P. Wegner. The Object-Oriented Classification Paradigm. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, pages 479 – 560. MIT Press, 1987.
31. J.F.H. Winkler and S. Kauer. Proving Assertions is also Useful. *SIGPLAN Notices*, 32(3):38 – 41, 1997.
32. XEROX Palo Alto Research Center, `http://aspectj.org`. *Homepage of AspectJ*, 2000.