Reasoning About a Classification of Crosscutting Concerns in Object-Oriented Systems

Constantinos A. Constantinides
Department of Computer Science and Information Systems
Birkbeck College, University of London
cc@dcs.bbk.ac.uk

Therapon Skotiniotis
College of Computer Science
Northeastern University
skotthe@ccs.neu.edu

Abstract

In this paper we discuss some issues regarding the nature and behavior of crosscutting concerns in OO systems. We argue that aspects must be defined as first-class abstractions in order to be manipulated as such and thus to provide for reusability and dynamic adaptability as well as for the creation of dynamically loadable aspect repositories.

1. INTRODUCTION

Object Orientation has provided the means to modularize data abstractions along with operations on them. It further provided mechanisms to group, reuse and extend behavior as well as internal component structure. Recent research has demonstrated that OOP gives rise to certain problematic issues. More specifically, inheritance anomalies [18] emerged from the discussions on conflicts between synchronization and reuse and the need for better separation between synchronization code and the main functionality in concurrent programs. Generalizing on inheritance anomalies, code tangling [16] has given rise to Aspect-Oriented Software Development (AOSD), with several technologies being developed to tackle this issue while forcing software engineers to revisit the notion of concerns and issues on their separation.

A component is a structure that addresses a concern. At the implementation level it is defined by a module with well-defined behavior and an interface that can be used as a building block for large software systems. A desired property, albeit not a necessary condition, includes plug-compatibility in order to minimize code duplication and maximize code reuse. An object is an example of a component and the dominant decomposition of Object-Oriented Analysis and Design (OOA/D) has stressed the importance of a component hierarchy. On the other hand there should, in our opinion, be a difference between an aspect at the conceptual level and one at the implementation level. Conceptually an aspect is a concern that affects the semantics or the performance of other concerns. At implementation level of traditional paradigms, an aspect cannot be localized in a single functional or reusable module. Different AOSD technologies support the notion of an aspect as a modular unit that encapsulates a related set of actions that take place at well-defined execution points throughout the base program referred to as joinpoints. Under AOSD, both design and implementation stages maintain a clear view of the concerns in the problem domain.

If developers view the system as a black-box during requirements and analysis, they will then focus on the main functionality of the system and up to that point decomposition would be functional and component-dominant, with aspects as second-class if present at all. Aspects would tend to arise more clearly during design and they will eventually have to be treated as equal with components in the sense that the eventual design class diagram will be built with no dominance of components over aspects. Further, the ability to provide component and aspect off-the-shelf (CAOTS) technologies can result in a high level of reuse and thus provide for a faster development, as well as for a high level of software maintenance.

In this paper, we discuss some issues regarding the nature and behavior of aspects, and we argue that it would be advantageous if aspects are defined as first-class abstractions in order to be able to be manipulated as such and thus provide for a high level of reusability and dynamic adaptability. This can also enable formal modelling with the support of UML. We further discuss issues that arise in order to obtain CAOTS.

2. SUPPORT FOR CAOTS

Although a clear distinction between a component and an aspect was provided in [12, 16], programming languages and tools deployed to enforce AOSD have yet to provide a mechanism (abstraction level) with which CAOTS can be achieved. We believe that a means to group (modularize), as well as to define the behavior of an aspect, will be a step in this direction for the following reasons: (1) modularization will allow for better abstraction and a higher level of reuse, (2) aspect types could then be defined to provide a means to characterize the intended behavior of aspects, and (3) at the moment technologies that support AOSD provide mechanisms to group (and / or organise) aspects depending on the concern that is addressed. For instance, in AspectJ [11] one can define abstract aspects from which new aspects can then be defined through inheritance, thus providing programmers with an aspect hierarchy that holds aspects of similar or sharing behavior. One would argue that the principle of substitutability should apply to aspects. The hierarchy of aspects, as specified by inheritance hierarchies, denotes subtyping thus supporting code reuse. The inheritance operation, as it is provided by some OO languages like Java, is therefore tangled with both concerns. Allowing separate definitions for behavioral subtyping and code reuse will allow for better separation at the language level. This separation could be then exploited in order to check that subtypes are used correctly by the runtime system.

The issue that we see here is that separation is simply the first step in a divide and conquer approach. Separation alone does not solve all our problems, but rather it refines the problem to smaller, more distinguishable and easily manageable parts. Attaching, or composing these units (components) together in the correct order for a specific problem, will yield the final semantically correct program for the task at hand. Thus, the ability of a language or tool to provide constructs to define each concern has to be complemented with its ability to further allow for the definition of aspect/component com-

position. We would also like for a language or tool to provide a flexible mechanism by which both aspects and components could be easily reused and composed together into a running program.

3. OPEN ISSUES IN AOSD TECHNOLOGIES

There are currently a number of different proposals that describe aspect-oriented software architectures. Although the mechanism of weaving lies at the center of their differences, they all have the same goals: Better abstraction than what has so far been achieved with current (OOP, or other) methodologies, higher modularity and higher degree of reuse. In this section, we would like to raise the following concerns: (1) Some of the open issues regarding the design and implementation of an aspect oriented architecture lie at the means to better express the aspectual properties of systems. Should one use an aspect language or a more general framework? What are the tradeoffs? (2) Another issue is the level at which aspects and components integrate. Should the integration be at the source code or at the object code? (3) To what degree should an aspect oriented architecture support an open system? Should it enable dynamic adaptability? (4) To what degree should reusability be supported? Do we want aspect instances to be made available in dynamically loadable libraries? and (5) Should an architecture further enable formal verification of system properties? Do we want our system to accept all dynamic attachments unconditionally or should a verification step be added to safeguard against inappropriate compositions (either due to an interface mismatch or failure to support pre- and postconditions in aspects or components).

3.1 Static vs. dynamic weaving

Aspect thinking has been traditionally present implicitly in software systems where aspects were manually woven into components (and into each other). One difference in the proposals for supporting AOSD resides in the way in which aspects are woven across the functional components of the system. How weaving will be achieved can depend on a number of factors. One factor is whether one would decide to provide linguistic support for aspect definition and the inlining of aspect code into functional code, or whether one would use other technologies such as reflection (to address aspect definitions on the meta-level) or even a more general framework approach.

A distinction can be made here between static and dynamic weaving. Static weaving implies that the code injected into the relative base system is solely dependent on the static properties of the base program. On the other hand dynamic weaving refers to the ability of the language or system to allow for runtime evaluation in order to resolve aspect execution point definitions (pointcuts) and the incorporation of advice at the resolved execution point, as in the case with dynamic joint point designators of AspectJ. For example, the code below

```
pointcut getters() :
call (public void getX()) && target(Reader)
```

specifies `getters()` to be a dynamic pointcut referring to all calls of `getX()` to objects of type `Reader`. AspectJ allows for both a method name as well as the target object to be determined at run-time through regular expressions and late binding respectively, thus providing for dynamic join points. The pointcut below

```
pointcut Mytracer() :
call( * A.get*(..) && this(B))
```

is the conjecture of all method calls with any return type, whose name matches the pattern `get*`, on object `A`, and the currently executing object is of type `B`. It essentially picks up all possible join points that satisfy both of the predicates as described above.

Static weaving (i.e. at source code ala AspectJ) brings about the advantages of comparable running time (to programs written without AOP) and also allowing for a dynamic join point model through the deployment of reflection / call graph analysis. On the down side though, the aspectual code (advice as well as pointcuts) are no longer distinguishable at run-time. Therefore aspects cannot be materialized at runtime making runtime reconfiguration hard (if not impossible). The extend to which a language supports join point definitions is implementation dependent. In AspectJ, for example, the flow graph allows for broad definition of join points. However, AspectJ does not support aspect instantiation or separate compilation. As a result, run-time adaptability is not supported. A feasible approach to handle dynamic weaving is to implement aspects using meta-objects. DJ-Aspects [21] is a proposal that addresses dynamic weaving through the use of metaprogramming. Other approaches include aspect-oriented frameworks like the AMF [2, 4, 5] that provides dynamic weaving in a framework. We believe that ideally an implementation should support both static and dynamic weaving.

3.2 Level of weaving and life span of aspects

The level of weaving defines the point up to which one manages (or wishes) to achieve separation of concerns in the system and it is also related to the notion of life span of aspects. The level of weaving moves along the compilation axis, and it can be pre-compile, compile or post-compile (execution time) weaving. When aspect code is inlined into component code before compilation, aspects do not exist at run-time. Reflective technologies such as DJ-Aspects address aspect definitions at meta-level. With reflective systems aspect code exist at the source as well as in the run-time system. Aspect-oriented frameworks also achieve separation of concerns until execution time.

3.3 Openness property: Static and dynamic adaptability

As requirements change and as slightly different problems arise over time, a system must be able to change and evolve. An essential requirement for complex systems is that they shall have an open software architecture by which non-invasive adaptability can be performed, i.e. concerns can be dropped, reconfigured or deployed and be connected with newly installed concerns, without in each case forcing reengineering of either the system itself or the client code. In the context of AOSD, adaptability is essentially a form of composition of concerns. Dynamic adaptability ensures that runtime requirements can be met without the need to halt the running system in order to ensure proper reconfiguration. Essential for the provision of dynamic behavior is the survival of aspects at run-time [17]. Recent work by [15] has further provided a formal approach, which allows for dynamic aspect adaptability to be realized from a static, type safe environment.

4. TOWARDS A CLASSIFICATION OF ASPECTS

A categorization of aspects can be beneficial towards the understanding of their nature. In the subsequent subsections a categorization of aspects is presented based on several design dimensions.

4.1 Functional vs. non-functional

Since the definition of an aspect is that it is a concern that produces tangling between two or more concerns, the nature of aspects does not confine them to being non-functional requirements. Further, aspects are concerns that are dependent on the problem domain. As a result, aspects may address either functional or non-functional requirements. Functional requirements address the behavior of the system in terms of the services it provides, whereas non-functional requirements attach constraints to these services, thus affecting the performance and semantics of the system. As an example, consider a GUI application that can display quadrilaterals. One would like to enhance the GUI program so that each drawn quadrilateral also has a title bar on its one side so that the user can minimize or maximize the image. Solutions to this problem vary, and one way to go about this problem is to extend the existing class (thus acquiring all its operations). This will allow the addition of the code responsible for creating the title bar in the extended class, while still having access to the parent's code [9, 10]. By capturing the extended operation of the title bar in an aspect one will simply weave the aspect at specific execution points within the program so that it is called when needed. Notice that the aspect-based solution to the problem does not require any changes to the clients that use the GUI application. The clients can still send their requests to the same objects. However, in the inheritance-based solution all clients that would like to use the new title bar feature need to know that there is a different class (the subclass) that implements this functionality so that they can now call the new extended class for the new service. Detailed discussion of this problem (otherwise known as the "extensibility problem") and its solution using units and mixins can be found in [10]. On the other hand, examples of non-functional requirements include synchronization, scheduling, authentication, tracing, and logging.

4.2 Component span

In an object-oriented system, aspects could be categorized according to their level of crosscutting in the class hierarchy of a system. This approach can identify intra-class (intra-object) or inter-class (inter-object) aspects. Naturally, the two categories are not mutually exclusive, as one aspect may cut across several methods of one class but at the same time it could cut across different objects. To this end it is necessary to distinguish between aspects at the conceptual level and at the implementation level.

Consider a collection of classes that provide different services in a concurrent system. At a conceptual level, synchronization is one possible aspect of concern. However, different technologies would address this aspect differently at the implementation level. AspectJ is able to provide one aspect definition that can address both intra-object and inter-object crosscutting. In the Composition Filters model [1], filters may address intra-object and inter-object concerns as well. DJ/DemeterJ [20] allow one to specify advice (through Visitor methods) that will take effect on the subset of classes that the traversal specification denotes. Traversal semantics being structure-shy [20], allow for adaptive programs that apply to a range of OO designs (i.e. class graphs). The AMF does not support inter-object crosscutting. On an intra-object level, the AMF supports one aspect definition for each method associated with this aspect.

4.3 Layer span

It is also important to distinguish between aspects that may exist at the system level and aspects that may exist at the application level while some aspects may cut-across these layers. An example of a multi-layer aspect is Quality of Service, which may span the application layer, the operating system layer, and the network layer as well. Another multi-layer aspect example would be encryption with the existence of different algorithms at each layer.

4.4 Support for assertions and Design by Contract

We can also view aspects with respect to the degree to which they can support Design by Contract (DBC) [19]. Under this principle, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of some mutual obligations (contracts). The concept of deploying assertions and DBC in the context of AOP was discussed in [2, 3, 13]. DBC was introduced in the context of the Eiffel programming language, where a contract is "hard-wired" in the class definition and it is inherited. Further, in iContract [14] assertions are propagated along inheritance and interface relationships. Separating contracts from the functional components could achieve a higher level of reusability and adaptability. In the case of concurrent behavior, this would avoid inheritance anomalies. Implementing DBC in an AOSD environment that will view aspects as first-class abstractions will enable a more flexible and dynamic contract deployment that will facilitate the separation of code reuse from behavior reuse during an inheritance operation. Currently no AOSD technology enforces DBC but the current implementation of AspectJ can support the definition of contracts as aspects through the implementation of pre- and post-conditions.

4.4.1 Contracts to verify functional components

When assertions are placed in aspect interfaces, component-aspect communication would then be based on well-defined protocols, thus observing the semantics of the system. Using contracts implemented as aspects, developers will be able to allow each program to define pre- and post-conditions as well as invariants. Contract definitions can then be checked at compile and/or runtime [9] to verify that the composition is valid and correct according to the contracts set forth by the developer.

In general we would like to have aspects to be concrete enough to be identified and managed at runtime as first-class entities, but also general enough to support obliviousness and quantification when it comes to AOP [8]. Further, we would like to stress the importance of aspects as modular and instantiable entities. This ontology of aspects allows for dynamic adaptability to be possible in the AMF. The fact that we can refer to aspects at run-time allows for the provision of dynamically reconfigurable systems.

4.4.2 Contracts to verify aspects

It is also important to note that we might also need to provide contracts for aspects in order to be able to enforce a runtime check of rules that will guard against an improper run-time deployment of aspects. This would be beneficial for architectures that support run-time adaptability. The reason behind this is that adaptability of aspects may imply alteration of data/execution flow of the base program, resulting in incorrect output or non-graceful termination of the system itself. Since aspects are deployed at specific execution points of the program, but also can be potentially used by other programs (if the concern being addressed is suitable to other problem domains) an issue arises as to when is an aspect "valid", or "correct" in the context of a given base program. Having the aspect code verifying its correctness with respect to the system it is about to be deployed at, it will provide more correct code and better integration of aspects and base systems. As a result, certain issues need to be addressed:

- What is the contract checking semantics between an aspect and a potential base system? The issue of aspect invariant needs to be addressed. One question is how do we specify an aspect invariant that has an "around" method (ala AspectJ "around" advice) which alters the return value of a method in the base system? In other words, how would one check and verify that the behavior of an aspect does not violate (falsify) the desired functionality of base systems. Do the solutions provided by the resulting system after the deployment of a new aspect still satisfy the initial requirements?
- The order of contract evaluation has to take into account the base system together with aspects that are currently active as well as new aspects that would possibly be deployed.
- How would we guarantee the truthfulness of the invariants of base programs? Aspectual behavior could potentially alter the behavior of methods or components. Therefore, aspects of this nature impose some new behavior that affects the functionality of the program. This added behavior should be checked in order to guarantee that the new added code still provides results within the solution domain that the overall system specification defines as acceptable. This checking mechanism will cut across many components at different points of execution. The checking technique in this case is thus more complex than what we would have faced with DBC in OOP. In other words, how would the system verify that aspects whose advice affect the behavior of programs still provide an acceptable semantics according to the system specification? Further, a runtime contract check mechanism is needed (along with a formal proof) that would also check contracts of dynamically added aspects to a system, addressing of course the issues mentioned above as well as issues of substitutability of subtypes (object subtypes or even aspect subtypes if such a categorization is developed). Therefore, a contract checking tool should be extended to take care of the validation of contract specification while at the same type maintaining the laws of subtype substitutability [9].

4.5 Relation to time: static and dynamic aspects

Another classification can be made according to the aspect-component relation over time as well the aspect policy over time. We can identify two cases under this approach: (1) A run-time association between components and aspects and (2) A dynamic change of aspect policy (Figure 1). Under (1), an aspect A_1 may cut across components C_1 and C_2 at time T_1, but then may cut across components C_1, C_2, C_3 at time T_2. As a result, the crosscutting was extended at run-time. Under (2) an aspect A_1 may cut across components C_1, and C_2 and C_3 at time T_2 with policy P_1, and then at time T_3 the aspect policy changes to P_2. In this case, policy change may require aspect run-time adaptability, i.e. a replacement of one aspect reference by another. Naturally, we may have a combination of (1) and (2). In both cases the examples refer to real-time environments and as such we imply that dynamic adaptability is supported by the underlying technology.

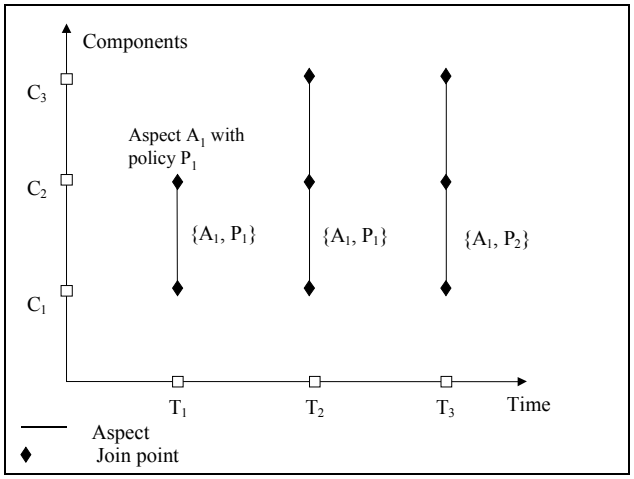

Figure 1. Aspect-component relations over time.

An aspect that is introduced at run-time and an aspect whose policy must change at run-time do not necessarily belong to the same category. In order to introduce aspects at run-time it requires the provision of an open system. If an aspect has been coded so that it will provide services (behavior) that would be state sensitive (either to system state or another aspect state) is a matter of the aspect (concern) itself. The ability of a system to allow for dynamic aspect deployment provides more flexibility and better software evolution since developers do not have to halt and alter existing aspects, but simply override or extend aspects in order to provide extra functionality / behavior. The reasons for deploying an aspect at runtime are related to system evolution, and they can be in order either to provide new semantics to the system due to a change in requirements, or to alter an existing system policy to meet some new requirements. Based on whether an aspect policy may have to adapt at run-time we can address the notion of static and dynamic aspects. An aspect such as authentication can be considered static, as it is very unlikely that an authentication policy will have to change during program execution. On the other hand, a scheduling policy will most likely have to be adapted at run-time. We can therefore view scheduling as a dynamic aspect.

Consider the example of an on-line conference room reservation system where we have a server providing View and Book services. The server would implement the readers-writers protocol to ensure fairness between readers (View) and writers (Book) or perhaps give a priority to writer client requests. The aspects of concern here are synchronization and scheduling. We might then decide to introduce a new service,

namely Cancel. Cancel is a 'write' operation. The introduction of Cancel will give rise to a race between Book and Cancel because our existing scheduling policy had no way of knowing that another 'writer' would have been introduced into the system. So not only scheduling and synchronization were introduced to a new component, but the scheduling policy that applies to existing components will now have to be modified to handle Book and Cancel, which are both 'writers'. In this example, one static aspect (synchronization) and one dynamic aspect (scheduling) were introduced at run-time.

Aspect languages modify the source code of a class by inserting aspect-specific statements at join points. The result is a highly optimized woven code whose execution speed is comparable to that of code written without AOSD. However, this makes it difficult to later identify aspect-specific statements in woven code. As a consequence, adapting aspects at run-time can be time consuming or not possible at all. On the other hand, aspect-oriented frameworks such as the proposal in [4] trade performance over the provision of dynamic adaptability.

4.6 Level of support for obliviousness

It would be interesting to investigate the level to which an aspect supports obliviousness [8] during adaptation time. Will (or should) the introduction of an aspect or a change of policy of an aspect require a modification of client code? Ideally, a system should support non-invasive dynamic adaptability and client code should be oblivious from aspect adaptation. To this effect, language mechanisms can be provided to validate aspect deployment semantics for static adaptability. Further, mechanisms such as contracts can be deployed in order to observe correct system semantics for dynamic adaptability.

4.7 Non-orthogonality

It is highly unlikely that aspects are completely independent of each other. Most real-life examples involve aspects that tend to be interdependent. This is termed non-orthogonality between aspects. The issue of non-orthogonality is important as it addresses the overall semantics of the system. Close to this is the issue of the order of activation of aspects. Consider a fine-grained concurrent system that supports the readers-writers protocol: the "before" behavior of a service dictates that synchronization should be evaluated before scheduling. If authentication is introduced, it has to be evaluated before synchronization. However, the "after" behavior of a service dictates that updating scheduling counters must be performed before synchronization.

```
// "before" behavior
preactivation(write):
<authentication, synchronization, scheduling>

// "after" behavior
postactivation(write):
<scheduling, synchronization, authentication>
```

The issue of non-orthogonality poses a number of interesting issues, as the developer should not be able to misuse the provision of dynamic adaptability and violate the semantics of the system. If two (or more) aspects are non-orthogonal, any action that is directed at one might affect the overall group. As a result, the system must be capable of preventing any possible misuse. In the example above, one should not be able to either drop synchronization or change the order of synchronization and scheduling. The order of aspect activation is supported in technologies such as AspectJ, Composition Filters and the AMF through different mechanisms.

4.8 Aspects as first-class abstractions

We think that the issue of what would constitute the most appropriate abstraction level at which to describe aspects still remains an open problem. Aspect languages provide constructs to express crosscutting concerns and a number of proposals support the implementation of aspects in a modular and natural way. There are proposals of specific languages for the support and implementation of AOP versus extensions to general-purpose languages. Domain-specific languages provide language constructs to address certain concerns. In ESP [6] the functional part is separated from the synchronization code, but it is still in the same class. The D framework [16] first introduced a complete separation between functional and synchronization code. However, aspects in D are not first-class abstractions, but statements that get inlined to the functional components by the weaver. Perhaps the most notable general-purpose aspect language is AspectJ that provides an extension to the Java language where aspects can be expressed in classes as separate modular units. Although AspectJ aspects are implemented in a similar fashion to classes, aspects are not instantiated. In order to maintain separation of concerns at all levels of the life cycle an aspect should be treated and remain a distinct artifact. Furthermore, reusability of aspects and their advice should also be possible as well as a notion of grouping related aspects together in some hierarchical structure. In an OO language these features are captured through classes, relations between classes (“has-a”), and the inheritance mechanism. It is thus only natural to consider aspects as first-class abstractions (classes) materialized as objects at runtime. This approach would provide the means to parameterize / group and reuse aspectual code through hierarchical structures as in OOP. We would like to be able to refer to and manipulate aspects at runtime as concrete, modular units.

4.9 Aspects as reusable pieces of software

Ideally, both components and aspects as well as architectural and design decisions should be reused. The importance of reuse lies on the fact that it can cut down costs, increase productivity and improve the quality of software. It has already been argued in the literature that the general feeling that OOP promotes reuse and expandability by its very nature is rather a misconception as none of these issues is enforced. Rather, a software system must be specifically designed for reuse and expandability [7]. A number of researchers are suggesting that the maximum potential for reuse is over the entire life cycle, where reuse is applied to each phase, and it must begin in the requirements phase. Another important issue is that of the verification of components and aspects in isolation from each other as well as to test and verify the collaboration of components and aspects. This would constitute an important phase in the design process. In order to achieve a high degree of aspect code reusability, we must investigate the degree to which aspects and components are separate from each other (as well as whether aspects are separate from each other). AspectJ aspects are not generic, as they reference the join-points of the class to which new semantics are introduced. Aspects in the AMF are more generic, as they have no visibility over the functional components but rely on local data. In

DJ/DemeterJ reusability is obtained through the structure-shy traversals. That is, the Strategy graph is concerned about the nodes that are explicitly referred to in the Strategy's specification (e.g. "from A to B via C"). As long as the class hierarchy and class relations provide some path satisfying that Strategy then one can achieve code reuse. The term reuse here has to do with whether the advice code introduced actually has some positive effect on the overall calculation of the solution one is trying to achieve. DJ/DemeterJ are "adaptive" since they provide strategies that map to a number of OO program solutions. This fact is supported by the structure-shy traversal (specified by the Strategy) that the tools perform.

4.10 Dynamically loadable aspect libraries

If we can have reusable aspects as first-class abstractions with flexible paramerization and/or contractual definitions (i.e. with pre-defined interfaces), it would perhaps be possible in the future to create aspect libraries that can be locally deployed or domain specific aspect repositories that can be remotely invoked. The provision of dynamically loadable libraries of aspects can prove advantageous for technologies that support dynamic adaptability.

5. CONCLUSION

In our view, aspects should be viewed along the same lines as components: stand alone entities that could be easily adapted and incorporated into different systems without the need of specifically altering the original entity (black box), but rather extending it to meet new requirements. As such, aspects will have a defined interface as well as semantics. Programmers can then reuse aspects in their programs by simply incorporating them into their system and providing the necessary hooks for them to inter-operate with their system. Component integration will thus define the semantics of the final program in terms of the semantic definition of components and their collaboration. To ensure correctness as well as a safe component substitution and component integration, one should also perform some sort of checking between aspects and the integrated system. The idea of contracts has been used effectively for OOP, and we believe it can be extended to address this problem by addressing some specific issues found in aspectual systems. The nature of contracts and contract checking has itself an aspectual nature, which can be exploited to provide an extended contract system. Aspects could be used to define the contract checking mechanism that could then be used and extended to address aspectual/component systems for correctness.

REFERENCES

1. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A., "Abstracting Object Interactions Using Composition Filters", ed. Guerraoui, R., Nierstrasz, O., and Riveill, M., Object-Based Distributed Programming, Volume 791 of Lecture Notes in Computer Science, pp. 152-184. SpringerVerlag, 1993.
2. Constantinides, C. A., Bader, A., Elrad, T., Fayad, M., and Netinant, P., "Designing an Aspect-Oriented Framework in an Object-Oriented Environment", ACM Computing Surveys Symposium on Application Frameworks, M.E. Fayad, Editor, Vol. 32, No. 1, March 2000.
3. Constantinides, C. A., and Elrad, T., "On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns", Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems. Minneapolis, Minnesota, USA. October 16, 2000.
4. Constantinides, C. A., Skotiniotis, T., and Elrad, T., "Providing Dynamic Adaptability in an Aspect-Oriented Framework", Proceedings of ECOOP 2001 Workshop on Advanced Separation of Concerns, June 15-17, 2001, Budapest, Hungary.
5. Constantinides, C. A., Elrad, T., and Fayad, M., "Extending the object model to provide explicit support for crosscutting concerns", International Journal of Software Practice and Experience. (To appear)
6. Dempsey, J., and Cahill, V., "Aspects of System Support for Distributed Computing", Proceedings of the ECOOP 1997 Workshop on Aspect-Oriented Programming. Jyväskylä, Finland, June 10, 1997.
7. Fayad, M., and Cline, M., "Aspects of Software Adaptability", Communications of the ACM, 1996.
8. Filman, R. E., and Friedman, D. P., "Aspect-Oriented Programming is quantification and obliviousness", Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems. Minneapolis, Minnesota, USA. October 16, 2000.
9. Findler, R. B., and Felleisen, M., "Contract Soundness for Object-Oriented Languages", Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001).
10. Findler, R. B., and Flatt, M., "Modular Object-Oriented Programming with Units and Mixins", Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), 34(1):94-104, 1999.
11. Kiczales, G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold, W. G., "An Overview of AspectJ", Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001), LNCS, Vol. 2072, pp. 327-355.
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irving, J., "Aspect Oriented Programming", ed. Aksit, M., and Matsuoka, S., Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1997), Vol. 1241, pp. 220-242, 1997.
13. Klaeren, H., Pulvermüller, E., Rashid, A., and Speck, A., "Aspect Composition Applying the Design by Contract Principle", Proceedings of 2nd International Symposium of Generative and Component-based Software Engineering (GCSE 2000), Springer, LNCS 2177, pp. 57-69.
14. Kramer, R., "iContract – The Java Design by Contract Tool", Proceedings of Tools USA, 1998.
15. Lämmel, R., "Semantics of Method Call Interception", Proceedings of the Workshop Aspect-Orientierung der GI-Fachgruppe 2.1.9 Objectorientierte Software-Entwicklung, 2-3 Mai 2001, Universität Paderborn.
16. Lopes, C. V., "D: A Language Framework for Distributed Programming", Ph.D. Thesis, Graduate School of the College of Computer Science, Northeastern University, Boston, Massachusetts, 1997.
17. Matthijs, F., Joosen, W., Vanhaute, B., Robben, B., and Verbaeten, P., "Aspects Should Not Die", Proceedings of the ECOOP 1997 Workshop on Aspect-Oriented Programming. Jyväskylä, Finland. June 10, 1997.
18. Matsuoka S., and Yonezawa, A., "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", ed. Agha, G., Wegner, P., and Yonezawa, A., Research Directions in Concurrent Object-Oriented Programming, pp. 107-150, The MIT Press, 1993.
19. Meyer, B., "Applying Design by Contract", IEEE Computer, pp. 40-52, October 1992.
20. Orleans, D., and Lieberherr, K., "DJ: Dynamic Adaptive Programming in Java", Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, 2001.
21. Pryor, J., and Bastán, N., "A Java Meta-Level Architecture for the Dynamic Handling of Aspects", Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA 2000).