A framework to address a two-dimensional composition of concerns

Constantinos A. Constantinides, Atef Bader, Tzilla Elrad

Concurrent Programming Research Center
Department of Computer Science
Illinois Institute of Technology
10 W. 31 St. Chicago IL 60616, U.S.A
+1 312 567.5150 (phone) +1 312 567.5067 (fax)
{conscon, elrad} @charlie.cns.iit.edu, bader@delta.csam.iit.edu
www.iit.edu/~concur

Abstract

Although not bound to programming only, Aspect-Oriented Programming is a paradigm proposal that retains the advantages of OOP and aims at avoiding the tyranny of dominant decomposition. The goal is to achieve an improved separation of concerns in both design, and implementation. Our work concentrates on the aspectual decomposition of concurrent object-oriented systems. We address composition of concerns by the aspect moderator class that coordinates the interaction of components and aspects while preserving the semantics of the overall system. Our design framework provides an adaptable model that allows for an open language where new aspects (specifications) can be added and their semantics can be delivered to the compiler through the moderator. In essence the aspect moderator is a program that extends the language itself.

Keywords: Aspect-Oriented Programming, Concurrent Programming, Open Implementation.

1. Introduction

There are numerous benefits from having an important concern of a software system being expressed well localized in a single code section. We can more easily understand it, analyze it, modify it, extend it, debug it, reuse it etc. The need for dealing with one important issue at a time was coined as the principle of separation of concerns [Dijkstra76]. To date, the primary idea for organizing software systems has been based on software decomposition, where a problem is broken down into sub-problems that can be addressed relatively independently. Software decomposition and programming languages have been mutually supportive. Current languages and paradigms support a number of modular representations such as procedures, and objects. They further support composition of modules into whole systems. At the same time many systems have properties that do not necessarily align with the functional components and cannot be localized to modular units. Example properties include performance optimizations, failure handling, synchronization, coordination and scheduling policies. Aspects are defined as properties that cut across groups of functional components. While these aspects can be thought about and analyzed relatively separately from the basic functionality, at the implementation level they must be combined together. Programming them manually into the system's functionality using current component-oriented languages results in aspects being spread throughout the code. This code tangling makes the source code difficult to develop, understand, and evolve by destroying modularity and reducing software quality [Lopes97, Kiczales et al. 97].

The major goals of Object-Oriented Programming (OOP) are abstraction, modularity, and code reuse. On the other hand, OOP provides only one dimension along which concerns can be separated. This was coined the "tyr-

anny of dominant decomposition" by [Ossher and Tarr 99]. Although not bound to programming only, Aspect-Oriented Programming (AOP) is a paradigm proposal that retains the advantages of OOP and aims at avoiding the tyranny of dominant decomposition. In [Mens et al. 97, Lorenz98] AOP is viewed as a general modeling mechanism, which applies to all phases of the life cycle of the software. In fact, [Lunau97, Tekinerdogan and Aksit 98] encourage aspectual decomposition from the very beginning. Current AOP approaches view components and aspects as two separate entities where the aspects are automatically weaved into the functional behavior of the system in order to produce the overall system. In [Mens et al. 97] it was argued that generalized procedure languages do not provide the right abstraction for the description of aspects. The importance of having appropriate languages for the expression of aspects was also addressed and it was argued that aspect languages make aspect code more concise and easier to understand. If aspects are expressed in domain-specific languages, one needs an aspect language for every type of aspect and an automatic weaver tool would implement one (or more) aspect languages. On the other hand [VanHilst97] views weaving as a more general process that corresponds to component composition rather than merging. In [Ossher and Tarr 99] the authors argue that AOP does not go far enough with respect to tackling the tyranny of dominant decomposition as AOP permits only one decomposition. Instead, the concept of multi-dimensional separation of concerns is proposed.

2. Statement of the problems

One difference in the proposals for supporting AOP resides in the way in which aspects are weaved across the functional components of the system: One issue is whether the weaving is static or dynamic. Example architectures that impose static weaving include D [Lopes97], AspectJ [Lopes and Kiczales 98], D^2-AL [Becker98], and IL [Berger et al. 98]. Other architectures that make use of reflective technologies allow dynamic weaving. Examples include Luthier-MOP [Pryor and Bastán 99] and AOP/ST [Böllert99]. Another issue is whether there is code transformation. Technologies that rely on automatic weavers produce code transformation. Examples include D, AspectJ, D^2-AL and IL. Reflective technologies will typically not have code transformation. Additionally there are proposals of specific languages for the support and implementation of AOP versus extensions to general-purpose languages. Examples of the former include COOL, RIDL [Lopes97], D^2-AL, IL and TyRuBa [DeVolder98]. Examples of the latter include AspectJ, Replication-Framework [Fabry98], JST [Seinturier99], Luthier-MOP and Kava [Welch and Stroud 99]. Further, there are differences in the level of abstraction of these implementations. We summarize a comparison of AOP technologies in tables 1 and 2. Our position is that automatic weaver implementations and aspect languages impose a number of restrictions which we discuss in this section.

2.1 Increased complexity of a general-purpose aspect language. Where a general-purpose aspect language is implemented, the introduction of new types of aspects will require the language to be extended in order to provide new constructs for their representations. As the number of these aspects increases, the complexity of (both implementation and use of) the language would also increase.

2.2 Expressiveness of general-purpose aspect language. Can a general-purpose aspect language really be general purpose? We yet have to see real examples of expressing different kinds of aspects. On the other hand, the language designers must have a grammar specification in advance. As it is impossible to predict the syntax of possible future aspects we argue that an aspect language can really be general purpose if it is constantly expanded.

2.3 Restrictions in specific aspect language implementations. [Beugnard99] argues that AspectJ imposes a restriction by enforcing the explicit reference to the code in the aspect, making it reusable only for that purpose. The author proposes a separation of the aspect description in two parts: the semantics of the aspect itself, and the join points.

2.4 Restrictions imposed by static weaving, and lack of support of dynamic weaving. In current implementations, the weaving process is static. Aspects reference the classes of those objects whose behavioral additions describe, and define the points at which additions should be made. Static weaving means to modify the source

code of a class by inserting aspect-specific statements at join points. In other words, aspect code is inlined into classes. The result is a highly optimized woven code whose execution speed is comparable to that of code written without AOP. However, static weaving makes it difficult to later identify aspect-specific statements in woven code. As a consequence, adapting or replacing aspects dynamically during run-time can be time consuming or not possible at all. An example where this would be beneficial is given by [Matthijs et al. 97] where a load balancing aspect could replace the load distribution strategy woven before with a better one depending on the current load of managed servers. Another example is given by [Böllert98b] where a particular malfunction in a software system should deploy a tracing aspect to be woven and run without having to restart the software system. Currently, automatic weaving is not a viable technology for implementing aspects in a dynamic environment. Static weaving has advantages over performance [Böllert99] whereas dynamic weaving facilitates incremental weaving and makes debugging easier. Ideally, an implementation should support both static and dynamic weaving. A feasible approach to handle dynamic weaving is to implement aspects using meta-objects. Proposals that address dynamic weaving through the use of meta-programming are [Lunau98], AOP/ST and Luthier-MOP.

ARCHITECTURE / FRAMEWORK	**DESCRIPTION**	**LANGUAGE(S)** **ADL: ASPECT DEFINITION LANGUAGE** **CL: COMPONENT LANGUAGE**	**IMPLEMENTATION**
D	Synchronization / Coordination architecture	COOL(synchronization ADL) RIDL (coordination ADL)	Automatic weaving
AspectJ	General purpose aspect language	Java extension (ADL) Java (CL)	Automatic weaving
D^2-AL	Aspect language for distribution control	D^2-AL (ADL) Java(CL)	Automatic weaving
IL	Aspect language for object interaction	IL (ADL) Smalltalk, Open-C++ (CL)	Automatic weaving
Replication Framework	Aspect language for replication. Aspect language for error handling	Java extension (ADL)	Automatic weaving
Extended Computational Reflective Architecture	Metaobject architecture		Reflection
TyRuBa	Simplified Prolog	COOL subset (ADL)	Automatic weaving
JST	Aspect language for synchronization	JST – Java extension (ADL) Java (CL)	Reflection: Weaver produces OpenJava meta-class
Luthier-MOP	Metaobject architecture	Extension to Smalltalk	Reflection
AOP/ST	Metaobject architecture	Smalltalk	Reflection
Kava	Metaobject architecture	Extension to Java	Reflection

Table 1. Description of AOP architectures and frameworks.

2.5 Increased complexity of an automatic weaver. The weaver confines itself to the nature of the aspects and the constructs provided by the aspect language(s) it implements. The weaver must contain a specific interpreter for each aspect description language. If a new type of aspect of concern were to be added to the model, the weaver would have to be modified (extended) to adapt to the new model. As different aspect description languages address different types of aspects, an increase in the number of aspect languages would result in an increase in the complexity of the weaver.

2.6 Aspect inter-relationships. An open problem remains the issue of aspect-aspect interaction. We can break this down into two areas: 1) relationship of orthogonal aspects and 2) the existence of non-orthogonal aspects.

The relationship of orthogonal aspects relies on their order of activation and by their validation and verification. Current aspect-oriented architectures have addressed models where aspects are orthogonal and therefore present a flat structure. There are cases where aspects are not orthogonal, but they can cut across each other. In these cases the overall structure is not flat but rather hierarchical. The issue of non-orthogonal aspects still remains an open problem, which we believe can be attacked in either of two ways: 1) by language design and 2) by implementation.

2.7 Debugging and level of weaving. In several implementations that use automatic weavers, weaving is done before compile time. Of great interest is the issue of debugging an aspect program. In proposals such as AspectJ, the only way of thinking about and therefore debugging a program is to examine the woven code. We argue that this should not be the case. Today a programmer debugs at the level of Java [Gosling et al. 96], and C++ [Stroustrup86] and not assembly code. The implementor should not be constrained by the requirement that there exists woven code that is readable by the programmer.

ARCHITECTURE/ FRAMEWORK	SOURCE CODE TRANSFORMATION	WEAVING	LEVEL OF WEAVING	REFERENCE
D	YES	Static	Pre-compile	[Lopes97]
AspectJ	YES	Static	Pre-compile	[Lopes and Kiczales 98]
D^2-AL	YES	Static	Pre-compile	[Becker98]
IL	YES	Static	Pre-compile	[Berger et al.98]
Replication Framework	YES	Static	Pre-compile	[Fabry98]
Extended Computational Reflective Architecture	NO	Dynamic	Run-time	[Lunau98]
TyRuBa	YES	Static	Pre-compile	[DeVolder98]
JST	YES	Static	Pre-compile	[Seinturier99]
Luthier-MOP	NO	Dynamic	Run-time	[Pryor and Bastán 99]
AOP/ST	NO	Dynamic	Development-time Run-time	[Böllert99]
Kava	NO	Dynamic	Pre-compile	[Welch and Stroud 99]

Table 2. A comparison of AOP implementations.

2.8 Lack of criteria of aspect ordering. How does an automatic weaver cope with multiple aspects that must be woven into the same class, or the same method? Does the ordering have to be done manually, or should the weaver sort aspects according to some criteria?

3. The aspect moderator framework

Our work concentrates in concurrent object-oriented programming. In its simplest form, we view a concurrent (shared) object as being decomposed into a set of abstractions that form a cluster of cooperating objects: a functional behavior, synchronization, and scheduling. The behavior of a concurrent object can be reused, or extended. We view synchronization and scheduling as aspects, and we focus on the relationships between these abstractions within the cluster. We can shift the responsibility of an automatic weaver to an object, the aspect moderator, that would coordinate aspects and components together (figure 1). A proxy object controls access to the functionality class. The proxy uses the moderator object to create and evaluate the aspects for every method of the functionality class. The moderator uses the aspect factory in order to create aspects (figure 2). Before invocation, the proxy calls the moderator to evaluate its associated aspect(s). The aspect moderator class should be extensible in order to make the overall system adaptable to addition of new aspects. We also believe that this approach provides the flexibility to the programmer to retain the definition of aspects by current programming languages. It also provides the basis for a design framework that would make use of patterns whose importance

within the AO technology was addressed in [Lorenz98]. The aspect moderator class defines the semantic interaction between the components and the aspects. Further, the semantics of the model define the order of activation of the aspects.

4. Architecture of the framework

A sequential object is comprised of functionality control and shared data. Access to this shared data is controlled by synchronization and scheduling abstractions. Synchronization controls enable or disable method invocations for selection. The synchronization abstraction is composed of guards and post-actions. During the precondition phase, guards will validate the synchronization conditions. In the post-condition phase, post-actions will update the synchronization variables. The scheduling abstraction allows the specification of scheduling restrictions and terminate-actions. At the pre-condition phase, scheduling restrictions use scheduling counters to form the scheduling condition for each method. At the post-condition phase, terminate actions update the scheduling counters. During the precondition phase, the synchronization constraints of the invoked method are evaluated. If the current synchronization condition evaluates to TRUE the scheduling constraints are then evaluated. After executing the precondition phase, the moderator will activate the method in the sequential object. During post-condition, synchronization variables and scheduling counters are updated upon method completion. We stress the fact that the activation order of the aspects is the most important part in order to verify the semantics of the system. Synchronization has to be verified before scheduling. A possible reverse in the order of activation may violate the semantics. There are other issues that might also be involved. If authentication is introduced to a shared object for example, it must be handled before synchronization.

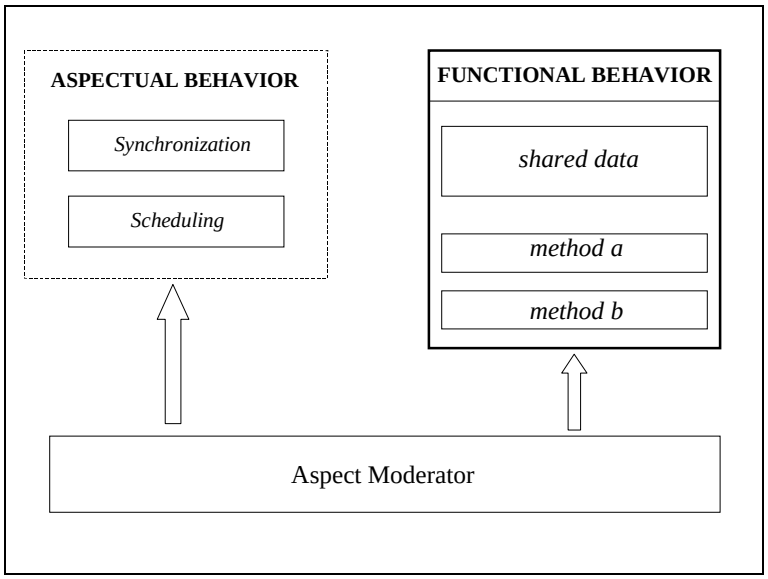

Figure 1. A concurrent object as a cluster of components and aspects.

4.1 The use of assertions to support software quality

A major component of quality in software is reliability: a system's ability to perform its job according to the specification (correctness) and to handle abnormal situations (robustness). [Meyer] introduces the concept of "design by contract" in the context of the Eiffel programming language [Meyer92]. Under this theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations known as contracts. These contracts govern the interaction of the element with the rest of the world. The importance of assertions is also stressed in [Jézéquel and Meyer 97] where it is described how the absence of specifications caused the disaster associated with the European Ariane 5

launcher. The aspect moderator framework adopts this approach in a different context: defining assertions (pre-conditions and post-conditions) as a set of design principles. Meyer argues that assertion monitoring yields to a productive approach to debugging, testing and quality assurance, in which the search for errors is not blind but based on consistency conditions provided by the developers themselves. As a result, reliability should be a built-in component in software development, not an afterthought. None of Java, Ada [DoD80] or CORBA [OMG98] has any built-in support for design by contract. In [Jézéquel and Meyer 97] the authors argue that without specification it is probably safer to redo rather than to reuse. Another important issue is the one of the verification of components and aspects in isolation from each other. One must be able to test the functionality of a component as well as being able to test that an aspect will align nicely with the functional components. Otherwise, there can be no guarantee that components and aspects will co-operate. In other words, one must test and verify the collaboration of components and aspects. This would constitute an important phase in the design process.

4.2 Adaptability

There is a general feeling that OOP promotes reuse and expandability by its very nature. We argue that this is a misconception as none of these issues is enforced. Rather, a software system must be specifically designed for reuse and expandability. In this framework both functional components and aspects are designed relatively separately from each other. This separation of concerns allows for reusability. Adaptability is an important quality factor in software systems. Incremental adaptability means coping with changing requirements without modifying previously defined software components. The conventional object-oriented model supports adaptability through composition, encapsulation, message passing and inheritance mechanisms. In general, lack of support of dynamic adaptability might lead to re-engineering the whole software system. In [Sanchez et al. 98] it is argued that concurrent OO languages do not provide enough support for the development of true adaptable software either because aspects are mixed in the functional components, or because once components are woven the resulting piece of software is too rigid to be adapted or reconfigured at run-time. The aspect moderator framework makes use of design patterns that hook components and aspects together, defining their semantic interaction. One of the advantages is that if a new aspect of concern would have to be added to the system, we do not need to modify the moderator class. We can simply create a new class to inherit and re-define it, and reuse it for a new behavior. The inherited class can handle all previous aspects, together with the newly added aspect. Adaptability is also applied to components. The aspect-moderator framework does not require some new syntactic structure for the representation of new aspects, but simply a new class for the new aspect. This technique makes it easy for an existing aspect to be removed from the overall system. In this framework, the moderator object has the capability to activate or drop aspects on the fly. Further, the semantic interaction between components and aspects in the framework is defined by a set of principles. Part of this semantic interaction is the order of activation of the aspects thus providing a criterion for aspect ordering. The order of execution can also be altered on the fly. This concept is not feasible with automatic weaver technologies. In this framework, components and aspects are designed relatively separately and they remain separate entities that may access each other freely without code transformation. In fact, functional components do not need to know about the aspect components in advance (before run-time) but only after an aspect has been created and registered by the moderator class. As a result, components and aspects discover each other at run-time if necessary. The interaction of newly added aspects with the rest of the system is handled in a similar manner as the implementor must specify the contract that binds a new aspect to the rest of the system rather than having to re-engineer the whole system. On the other hand, automatic weavers must rely on language constructs that are hard coded into aspect code to provide the contact (join) points. In [Matthijs et al. 97] the authors stressed the importance of aspect manifestation in every stage of development. The issue that in some cases aspects should remain run-time entities was also discussed in [Kenens et al. 98]. [Böllert98a] also stressed this issue by arguing that much like conditional compilation, aspects must be woven to the program on-demand. In technologies that rely on automatic weaving, aspects manifest in the model and in the program code, but neither in object code (byte code in the case of Java) nor in executable (binary) code. [Böllert99] argues that with static weaving it might be impossible to adapting or replacing aspects dynamically. The framework manages to achieve the manifestation of aspects at run-time. We argue that is important that in order to achieve maximum flexibility a framework must provide for dynamic aspect

evolution and ideally support both static and dynamic behavior. Based on run-time information, an aspect such as scheduling or load balancing might need to adapt itself. On the other hand an aspect such as synchronization can be statically dealt with.

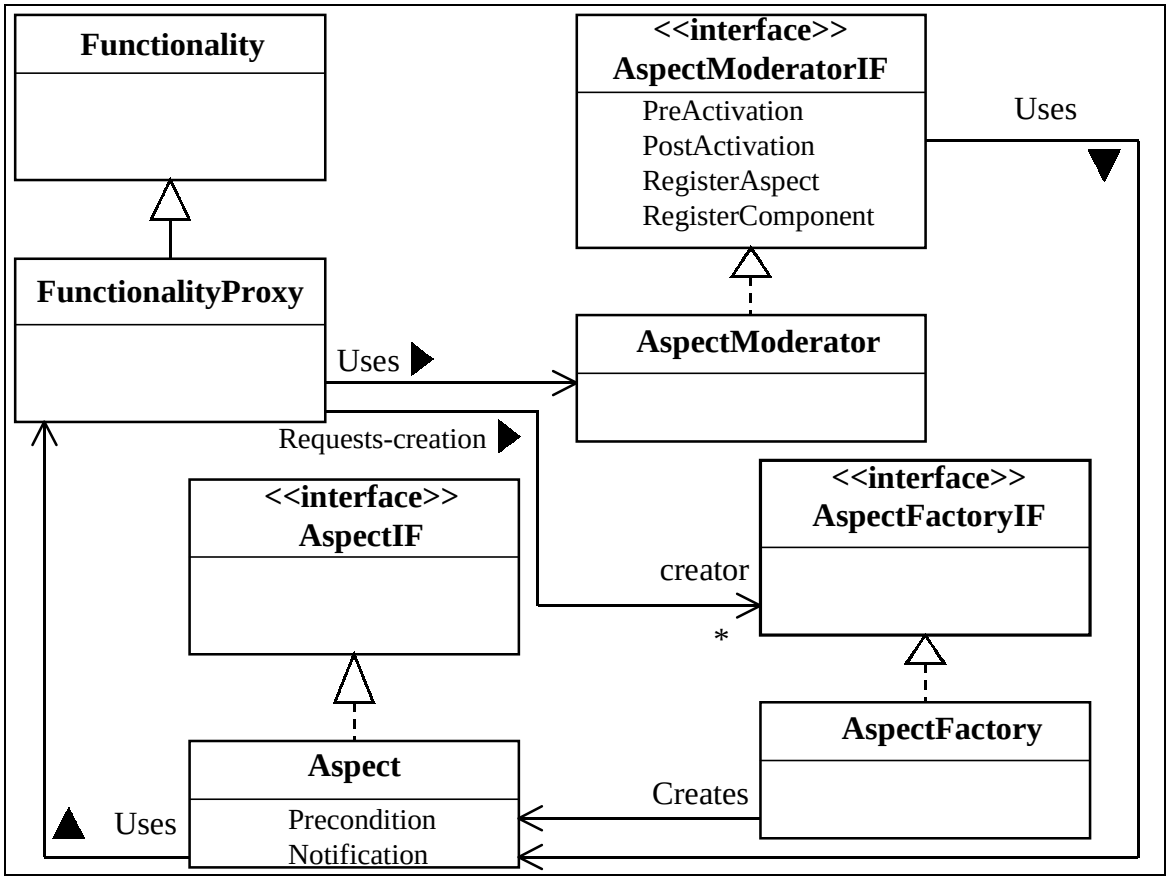

Figure 2. The architecture of the aspect moderator framework.

4.3 Composition of aspects

In ESP [Dempsey and Cahill 97] and the Adaptive Arena [Bader and Elrad 98b] the functional part of a system is separated from the synchronization code, but it still remains in the same class. The separation of functional and aspectual code in the aspect moderator framework results in program code that is more modular. Furthermore, the framework follows a general-purpose approach in order to address composition of concerns. This way, it is not confined to certain aspects but can address a number of aspects. It is also language neutral. With the exception of AspectJ, current technologies are confined in domain specific languages. We introduce the concept of an aspect bank, where the moderator of a cluster may initially need to collect all the required aspects from. The aspect bank makes use of the factory method pattern in order to create aspects although we view the bank more than just an aspect creator. The aspect bank provides a hierarchical two-dimensional composition of the system in terms of aspects and components.

5. Comparisons with current technologies

The framework puts the system under one compilation phase where an executable code is produced. Intermingled code exists only at the binary (executable) level. On the other hand, technologies such as AspectJ require two phases of compilation, one for the weaver to produce an intermingled source code and another for the final

compilation into an executable code. The level of weaving defines the point up to which one manages to achieve separation of concerns in the software system.

Both automatic weaver and the aspect moderator approaches provide the elegance of the original clean code during the analysis and design of the system.

The very reason that guided research towards AOP has resulted in the avoidance of the problem of inheritance anomaly. With the aspect moderator framework we manage to avoid the problem of inheritance anomaly since components and aspects are pure objects and can be therefore re-used. This problem is solved in automatic weaver technologies as well.

The concurrency facilities of the Java language provide a good choice to demonstrate the framework implementation but as AOP is not and should not be restricted to programming only (and thus not restricted to one paradigm or a particular language) the framework manages to remain language neutral and work is under way to identify other candidate languages. Particularly advantageous is the ability to express components and aspects in the same language as large-scale software systems are built based on COTS technology rather than domain specific languages.

A comparison between this framework and AspectJ is essentially a demonstration of the tradeoffs between a language and a framework. A language is ready to program but it is limited to the facilities that it provides. This framework can be viewed as an open implementation since the moderator provides a mechanism to support an open language. On one hand, a language implementor can always hard code a set of constructs to support a number of pre-defined aspects. Perhaps it would be impossible to predict all possible aspects that might come up and it would thus be impossible to predict their syntax and semantics. A language implementor would need to have the syntax in advance. On the other hand the framework provides a general aspectual capability to the system which is independent of a language. The aspect moderator is an architecture that allows for an open language where new aspects (specifications) can be added and their semantics can be delivered to the compiler through the moderator. In essence the moderator is a program that extends the language itself. Our approach has a good chance to reduce possible inconsistencies, although it cannot guarantee correctness.

6. Conclusion

We believe that AOP should be considered a discipline for general programming and should not confine itself in one application or a range of applications. It should not confine itself in a domain-specific language either. In this position paper we presented some preliminary work on an aspect-oriented framework for concurrent object-oriented systems. The overall behavior is made up of a functional behavior, concurrency aspects and a moderator class that coordinates the interaction between components and aspects while observing the overall semantics. Our approach partitions a system into a collection of cooperating classes in order to promote code reusability and make it easier to validate the design and correctness of these systems. This framework can provide for an adaptable model with ease of modification. The framework approach is promising, as it seems to be able to address a large number of aspects (and applications) as long as the inter-relationships of components and aspects (as well as the aspect-to-aspect relationships) are cleanly defined. A clean definition of these inter-relationships is achieved through the use of pre-conditions and post-conditions. In general we argue that a framework has a longer life span than a language (one that is not constantly extended). Further, we believe that a large language is generally undesired. A framework can therefore be viewed as providing a mechanism to address future needs with the minimum cost (in regards to time, financial and complexity cost). Clearly opening a language can be considered a risky approach, as the semantics of the extension mechanisms should balance openness with protection and security. In our framework the introduction of a new specification (aspect) must be accompanied by a set of rules that will ensure the integrity of the semantics of the system. These rules are expressed as pre-conditions, post-conditions, and order of activation of aspects.

7. References

[Aksit96] Mehmet Aksit. Composition and Separation of Concerns in the Object-Oriented Model. In ACM Computing Surveys. 28A(4). December 1996.

[Aksit97] Mehmet Aksit. Issues in Aspect-Oriented Software Development. Position paper at the ECOOP '97 workshop on Aspect-Oriented Programming.

[Bader and Elrad 98a] Atef Bader and Tzilla Elrad. Framework and Design Pattern for Concurrent Passive Objects. In Proceedings of IASTED/SE '98.

[Bader and Elrad 98b] Atef Bader and Tzilla Elrad. The Adaptive Arena: language Constructs and Architecural Abstractions for Concurrent Object-Oriented Systems. In Proceedings of ICPADS '98.

[Bardou98] Daniel Bardou. Roles, Subjects, and Aspects. How do they Relate? Position paper at the ECOOP '98 workshop on Aspect-Oriented Programming.

[Becker98] Ulrich Becker. D^2AL: A Design-based Aspect language for Distribution Control. Position paper at the ECOOP '98 workshop on Aspect-Oriented Programming.

[Berger et al. 98] L. Berger, A. M. Dery and M. Fornarino. Interactions between objects: an aspect of object-oriented languages. Position paper at the ECOOP '98 Workshop on Aspect-Oriented Programming.

[Beugnard99] Antoine Beugnard. How to Make Aspects Reusable; A proposition. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming.

[Böllert98a] Kai Böllert. Aspect-Oriented Programming Case Study: System Management Application. Position paper at the ECOOP '98 workshop on Aspect-Oriented Programming.

[Böllert98b] Kai Böllert. Aspect-Oriented Programming. Case Study: System Management Application. Graduation Thesis presented to the Fachhochschule Flensburg (Germany), 1998.

[Böllert99] Kai Böllert. On Weaving Aspects. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming.

[Constantinides et al. 99] Constantinos Constantinides, Atef Bader, Tzilla Elrad. An Aspect-Oriented Design Framework for Concurrent Systems. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming.

[Dempsey and Cahill 97] John Dempsey and Vinny Cahill. Aspects of System Support for Distributed Computing. Position paper at the ECOOP '97 Workshop on Aspect-Oriented Programming.

[DoD80] U.S Department of Defense. Ada Reference Manual. July, 1980.

[Dijkstra76] Edsger W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.

[Fabry98] Johan Fabry. Replication as an Aspect. Position paper at the ECOOP'98 workshop on Aspect-Oriented Programming.

[Gosling et al. 96] J. Gosling, B. Joy, and G. Steele. The Java™ Language Specification. Addison-Wesley, ISBN 0-201-663451-1, 1996.

[VanHilst97] Michael VanHilst. Subcomponent Decomposition as a Form of Aspect-Oriented Programming. Position paper at the ECOOP ’97 workshop on Aspect-Oriented Programming.

[Jézéquel and Meyer 97] Jean-Marc Jézéquel, and Bertrand Meyer. Design by Contract: The Lessons of Ariane. In IEEE Computer. January 1997, pp. 129-130.

[Kenens et al. 98] 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. Position paper to the ECOOP ’98 workshop on Aspect-Oriented Programming.

[Kiczales et al. 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Proceedings of ECOOP ’97. LNCS 1241. Springer-Verlag, pp. 220-242. 1997.

[Lamping97] John Lamping. The Interaction of Components and Aspects. Position paper at the ECOOP ’97 workshop on Aspect-Oriented Programming.

[Lopes97] Cristina V. Lopes. D: A language Framework for Distributed Programming. Ph.D. Thesis. Graduate School of the College of Computer Science. Northeastern University. Boston, Massachusetts, 1997.

[Lopes and Kiczales 98] Cristina Lopes and Gregor Kiczales. Recent Developments in AspectJ. Position paper at the ECOOP ’98 workshop on Aspect-Oriented Programming.

[Lorenz98] David H. Lorenz. Visitor Beans; An Aspect-Oriented Pattern. Position paper in ECOOP ’98 workshop on Aspect-Oriented Programming.

[Lunau97] Charlotte Pii Lunau. A Reflective Architecture for process Control Applications. In Proceedings of ECOOP ’97. Lecture Notes in Computer Science 1241. Pages 170-190. Springer-Verlag, 1997.

[Lunau98] Charlotte Pii Lunau. Is Composition of Metaobjects = Aspect Oriented Programming. Position paper at the ECOOP ’98 workshop on Aspect-Oriented Programming.

[Matthijs et al. 97] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pieere Verbaeten. Aspects Should not Die. Position paper at the ECOOP ’97 workshop on Aspect-Oriented Programming.

[Mens et al. 97] Kim Mens, Cristina Lopes, Badir Tekinerdogan, and Gregor Kiczales. Aspect Oriented Programming; Workshop Report. Position paper at the ECOOP ’97 workshop on Aspect-Oriented Programming.

[Meyer] Bertrand Meyer. Building bug-free O-O software: An introduction to design by contract™. Available electronically at http://www.eiffel.com/doc/manuals/technology/contract/page.htm

[Meyer92] Bertrand Meyer. Eiffel: The Language. Prentice-Hall, 1992.

[OMG98] Object Management Group. The Common Object Request Broker: Architecture and Specification. 1998.

[Ossher and Tarr 99] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns in Hyperspace. Position paper at the ECOOP ’99 workshop on Aspect-Oriented Programming.

[Pryor and Bastán 99] Jane Pryor and Natalio Bastán. A Reflective Architecture for the Support of Aspect-Oriented Programming in Smalltalk. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming.

[Sanchez et al. 98] Fernando Sanchez, Juan Hernandez, Juan Manuel Murillo, and Enrique Pedraza. Run-Time Adaptability of Synchronization Policies in Concurrent Object-Oriented Languages. Position paper at the ECOOP '98 workshop on Aspect-Oriented Programming.

[Seinturier99] Lionel Seinturier. JST: An Object Synchronization Aspect for Java. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming.

[Stroustrup86] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1986.

[Tekinerdogan and Aksit 98] Bedir Tekinerdogan and Mehmet Aksit. Deriving Design Aspects from Canonical Models. Position paper at the ECOOP '98 workshop on Aspect-Oriented Programming.

[DeVolder98] Kris De Volder. Aspect-Oriented Logic Meta Programming. Position paper at the ECOOP '98 workshop on Aspect-Oriented Programming.

[Welch and Stroud 99] Ian Welch and Robert Stroud. Load-time application of aspects to Java COTS Software. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming.