A Model for Describing Object-Oriented Systems from Multiple Perspectives

Torsten Nelson, Donald Cowan, and Paulo Alencar

Computer Systems Group, University of Waterloo
{torsten,dcowan,alencar}@csg.uwaterloo.ca

Abstract. We present work on a formal model for the composition of object-oriented modules, or *hyperslices*, which represent different perspectives on the system being built. With the model, we should be able to study existing approaches such as subject-oriented programming, as well as extend other object-oriented languages, such as the UML, to accommodate the use of hyperslices. We show here a sample of the specification language that accompanies the formal model, and a small example of its use.

1 Introduction

The idea of dividing a system description into various perspectives, each of which portrays some aspect of the system, has been the topic of recent intensive research. Techniques such as subject-oriented programming [9] and aspect-oriented programming [14] explore the possibilities of having multiple representations of the same entity from the problem domain. The need arises for a formal model that brings together what is common about the various multiple-perspective methods, particularly within the object-oriented paradigm. Such a model would allow for strengthening of existing techniques with the possibility for formal proofs and consistency checks, and serve as the basis for extending popular design and implementation languages with multiple-perspective capabilities. This work attempts to fill this gap.

In section 2 we discuss the need for multiple-perspective decomposition and present its different categories. We then discuss issues related to multiple-perspective decomposition restricted to the object-oriented paradigm in section 3. Section 4 shows the proposed model, followed by a small example of its use in section 5.

2 Multiple-Perspective Decomposition

One of the most widely accepted tenets of software development is the fact that complex problems must be divided into smaller parts that are easier to understand and work with. Traditional approaches to modeling such as structured analysis and object orientation take a top-down approach, where a system is decomposed into several parts, each of which is again decomposed until each part is simple and cohesive. The parts are then implemented independently and composed to form the desired software system. The exact nature of the entities that form each part varies between approaches, but typically

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 237–248, 2000.

each unit of the decomposition is encapsulated in a procedure (*functional* decomposition), module, or class (*object* decomposition). It is common to think of each of these units as pieces in a larger puzzle, all of which fit together to form the solution.

Recently, new approaches to problem decomposition have been the subject of research. It is often useful to view the entire problem from a single perspective, discarding all the problem elements that are not relevant to the perspective, and thereby reducing problem complexity. The problem then becomes easier to understand and manage than when viewed in its entirety. In contrast to the top-down approach for functional decomposition mentioned in the previous paragraph, each perspective can be thought of as a glass sheet with some aspect of a painting: one sheet may have a black outline of the painting; another, its colors; others may have shadings, textures, and so on. Superimposing all glass sheets forms the complete painting.

What distinguishes this "multiple-perspective" decomposition from other forms of divide-and-conquer is the fact that the decomposed parts are not disjoint. In other approaches to decomposition, any entity from the problem domain appears in only one of the pieces after decomposition - no entity appears in more than one piece. By contrast, an entity may appear in any number of perspectives, and its definition can be different in each perspective.

We frame our concept of perspective in software engineering rather broadly using the following two definitions.

- A description of a software system has *multiple perspectives* if it is explicitly divided into two or more partial descriptions, and there are entities from the problem domain that appear in more than one partial description.
- A *method* uses multiple perspectives if it enforces the description of software with multiple perspectives and provides rules governing how perspectives are related to one another.

These definitions allow a wide variety of methods to be thought of as using multiple perspectives. We divide them into three categories according to the kind of partial description used: architectural, domain-specific, and user-defined perspectives. Our research focuses on methods that allow user-defined perspectives. In order to clarify the relationships among the proposed models, we discuss the three categories of multiple-perspective methods.

2.1 Architectural Perspectives

Structured analysis methods such as the one proposed by Gane and Sarson [8] use two different diagrams to describe a system: a module hierarchy chart describing the division of the system into modules, and a data flow diagram describing how data was transferred between the modules. Each diagram represents a partial description of the system, and contains representations of the same entities from the problem domain, and therefore fit our definition of a perspective. Modern object-oriented analysis and design methods such as the Object Modeling Technique (OMT) [18], Coad and Yourdon [5], UML [3], and others, all use different perspectives to describe a software system.

However, in all of the above, the perspectives are defined by the method and reflect different modeling or descriptive capabilities. For instance, a model may define

structural relationships between entities, such as containment or association, while another may define behavioral aspects of each entity. Each perspective is usually described using a different language. This kind of description is often called an *architectural* perspective since each represents a certain aspect of the system's architecture. An important characteristic of architectural perspective decomposition is that it is independent of the problem domain. Dividing a problem into structural and behavioral views tells you nothing about the problem itself [13].

2.2 Domain-Specific Perspectives

There are other methods that allow the use of predefined perspectives, but instead of reflecting architectural aspects, the perspectives represent aspects of a specific problem domain. An example of this kind of method is the ISO standard for Open Distributed Processing (ODP) [11], which is geared towards the description of distributed systems. In ODP, systems may be described from any of five viewpoints. The five were chosen because they represent different and important perspectives relevant to the domain of distributed systems. These are *domain-specific* perspectives.

2.3 User-Defined Perspectives (Hyperslices)

Other methods allow for the decomposition of a problem into *user-defined* perspectives. These depend on the domain of concern, and therefore may be different from problem to problem. Also, the number of perspectives is not limited or predefined by the method. Methods supporting this type of decomposition were first developed for use in requirements elicitation, where it is common to have different stakeholders who have different ideas about the problem at hand, all of which must be captured. All perspectives are usually, but not always, described using the same language.

Various approaches have been proposed that allow user-defined perspectives. The approaches work at various levels of abstraction, and give different names to the units of decomposition. Table 1 lists some of these approaches.

Approach	Unit name	Level of abstraction
ViewPoints [7]	viewpoint	requirements analysis
Role models [15]	role	design
Contracts [10]	contract	design
Aspect-oriented programming [14]	aspect	implementation
Subject-oriented programming [9]	subject	implementation

Table 1. Approaches to decomposition with multiple user-defined perspectives

As can be seen from the table, the field is quite overloaded with different terminology for similar concepts. In recent work, Tarr et al. have coined the terms *Multi-Dimensional Separation of Concerns (MDSC)* to designate the discipline of multiple-perspective decomposition, and *hyperslice* to designate a unit of decomposition that follows this discipline, at any level of abstraction [20].

The most appropriate definition of hyperslice for our purposes is Daniel Jackson's concept of *view*:

> "A view is a partial specification of the whole program, in contrast to a module, which is a specification - often complete - of only part of the program" [12].

A hyperslice, like a "view", specifies some aspect of the entire program.

The form of decomposition afforded by hyperslices is useful in many levels of software design. During requirements elicitation, for instance, the engineer often has to deal with many different people who are stakeholders in the system to be developed. Each of these stakeholders may have different perceptions about the problem at hand and of the desired behavior of the system. Hyperslices are a natural way of representing each stakeholder's perspective on the system so that later a coherent picture can be formed. The same advantages found at the requirements analysis phase carry over into later phases. Having design and implementation modules that correspond to user requirements provides easy requirements tracing. In the next section we discuss the use of hyperslices with object-oriented systems.

While the ideas behind MDSC are useful, there are issues that must be tackled when applying it, One of the most pressing of which is is *consistency*. When we allow more than one representation of an entity, we must somehow ensure that the representations do not contradict each other. Being able to detect inconsistencies and deal with them is one of the challenges of MDSC, and formal models that allow reasoning are an indispensable aid towards achieving this goal. Easterbrook et al. have studied mechanisms for dealing with inconsistencies when using requirements-level hyperslices [6].

There are also conflicts that do not manifest themselves as logical inconsistencies, but as a form of interference between hyperslices; the goals of the viewpoints may be mutually interdependent and actions taken by one may cause undesired behavior in the other. This is analogous to the problem of feature interaction in telephone systems [4]. A formal model should aid in the detection of such interactions.

There is yet no formal model to allow a comparative study of the properties of the various existing approaches to hyperslices; our work intends to fill that gap. However, mention should be made of *Ku* [19], a formal object-oriented language for hyperslices currently under development at Imperial College. Once our model and Ku are both complete, it will be interesting to compare analytical results derived from the two independent studies.

3 Object-Oriented Hyperslices

While the concepts behind MDSC (Multi-Dimensional Separation of Concerns) are widely applicable, much of the work in the field involves its use together with the object-oriented paradigm. Many of the shortcomings of object-orientation are addressed by MDSC. Among these are rigid classification hierarchies, scattering of requirements across classes, and tangling of aspects related to various requirements in a single class or module. For a detailed treatment and motivation for the use of multiple perspectives in object-orientation, we refer the reader to Tarr et al. [20].

We are interested in the subset of MDSC that deals with object-oriented systems. We consider a hyperslice to be a module that conforms to some accepted model of object-orientation, made up of classes and class relationships such as containment and inheritance. For an object-oriented system to fit into the category of MDSC, however, there must be entities from the problem domain that appear in more than one module. That is, there must be classes in different modules that represent separate aspects of the same entity. Since a hyperslice is meant to be a complete encapsulation of some relevant aspect of the system, the classes in a hyperslice should not have any links to classes of other hyperslices. Each hyperslice should be a well formed unit that can be understood in isolation.

As an example, one of the hyperslices in a system may be concerned with displaying information about the various objects that concern the system. The classes in this hyperslice should have methods to output information, and whatever attributes that are relevant to these methods. Other attributes or methods, such as those concerned with synchronization, or with some form of computation over the data, should not appear in this hyperslice. However, the hyperslice should contain all classes that have an output aspect to them.

Splitting an object-oriented description into various hyperslices gives a level of separation of concerns that offers significant advantages over traditional approaches. Among these advantages are the following:

Requirements-based modularization since entities are allowed to appear in more than one unit of decomposition, and to be only partially specified in any one unit, it is possible to have units that encapsulate a single requirement from the problem domain. For instance, if some of the data is to be stored and retrieved from a network server, a hyperslice would be defined containing all classes that have data belonging in the server. Each class would have only enough detail defined to allow the relevant storage operations to be performed. The same classes might appear in other hyperslices, but no mention of server storage would be found in them.

Decentralized development since classes can be represented differently in different hyperslices, classes need no longer be owned by developers who are responsible for all details regarding that class, but each developer can be responsible for a part of the shared class.

Unanticipated composition designs for separate programs can be thought of as hyperslices of an integrated application; a hyperslice-enabled method would allow the developer to describe how the programs are to be joined together.

Software evolution along the same lines, features can be added to existing systems as separate hyperslices; this would allow changes to be made without requiring modifications to existing designs. An example of hyperslices applied to evolution can be found in [2].

3.1 Composing Hyperslices

Since each hyperslice is a plain object-oriented module, it can in theory be described using any object-oriented language, at any level of abstraction. Having defined the hyperslices, they must now be composed to form a complete system. This is where the

MDSC paradigm differs from other approaches to decomposition. Some approaches, such as that advocated by module interconnection languages, define interfaces for modules, with provided and required functionality, and match provided functions with required ones across modules. Others, such as frameworks, use inheritance as the basic composition mechanism. In MDSC, each hyperslice is well formed and independent, and does not require other hyperslices. There are classes, however, that exist in various hyperslices. In order to form the desired system, we must establish the *correspondence* between these classes.

Correspondence is the specification of what elements match between hyperslices, and the semantics of each match. There are many ways in which matching can affect the overall behavior of the system. Matched classes may have complementing behavior, or one's behavior may override the other, or they may interact in more complex ways.

The granularity of correspondence is an issue. Using classes as the unit of correspondence (also called *join point* [14]) seems to be too coarse. There are many different ways in which we may wish to specify that entire classes are matched, and the model would require a large variety of different correspondence operators. On the other hand, using single program statements is too fine-grained. Specifying correspondence would require understanding implementation details, and would be very complex. In our model, we choose the middle road and use methods as the smallest elements that can be matched. Ossher and Tarr argue in favor of this approach [16]. We define correspondence operators that allow methods to be matched with various different semantics. The semantics describe what method bodies are executed in response to a method call.

Another issue with regard to correspondence is *object binding*. This is the specification of how objects of classes with corresponding methods are bound to each other at runtime. The description of correspondence is class-based, meaning that a method from a class is said to correspond to a method in another class. However, the semantics of correspondence are observed during method invocation on particular runtime objects. If a method invocation results in the execution of a method in an object of a different class, we must be able to determine exactly what the target object is. The binding specification describes how to determine correspondence between runtime objects.

4 The Hyperslice Model

We are interested in studying object-oriented hyperslices with operation-level correspondence. A language to allow hyperslice composition is under development. This is a class-based language that uses methods as the smallest indivisible unit. It is not meant to be an executable object-oriented language, but a means to specify formally compositions that can be done in any object-oriented design or implementation language. The language has two parts, one for class definition and one for composition.

The model allows us to study properties of techniques that use operation-level correspondence such as subject-oriented programming [9], and serves as a semantic basis for the extension of other object-oriented languages with the required mechanism for multi-dimensional separation of concerns. Since subject-oriented programming already represents an embodiment of MDSC at the implementation level, one of our interests is in extending design-level languages such as UML.

4.1 Class Language

The class language defines the classes present in the system, as well as the call graph between the methods of these classes. The semantics of correspondence are specified in terms of transformations in the call graph structure.

Each class is defined as a set of methods. A method has a name and a list of other methods that it calls. The internal state of a method is irrelevant. Each method call in this list is decorated with the modal symbols □ (*always*) and ◇ (*possibly*). At this stage, the language is untyped, and does not support parameters to methods, or return values.

Data attributes are not modeled. Instead, they may be represented as methods, the data being the internal state of the method. In fact, any method that does not call any other methods can be thought of as an attribute. This approach is related to the one used by Abadi and Cardelli in their object calculus [1]. Figure 1 shows the syntax of the class language, while figure 2 shows a sample class definition.

$hyperslice$ ::= $class\{class\}$
$class$ ::= **class** $name$ "{" $method\{, method\}$ "}"
$method$ ::= $name[$ **"("** { *called-method* } **")"**]
called-method ::= $\Box name | \Diamond name$
name ::= an identifier containing letters, hyphens, or a period. The period separates class names from method names.

Fig. 1. Class definition language syntax

```
class Tree {nodes, find( □ nodes, ◇ travel-left, ◇ travel-right ),
            travel-left (□ nodes), travel-right (□ nodes) }
```

Fig. 2. Sample class definition

Note that the names inside the parenthesis are not arguments, but the list of methods that the method calls. In the example above, the class named *Tree* has four methods. The methods *find*, *travel-left*, and *travel-right* will always call method *nodes*. The method *find* may, in addition, call methods *travel-left* and *travel-right*. The method call list may contain methods from other classes, specified by stating the class and method names separated by a period (as in `Tree.find`).

4.2 Composition Language

A composition is specified using *calling contexts*. At the highest level is the program calling context. By default, in any context, calling a method results in that method being executed. Method correspondence expressions can be used to change the effects of method calls. An expression is formed by two method names connected by a correspondence operator. Each expression can also introduce new calling contexts that have scope limited to the execution of the method that precedes the context.

Operator	Call	Execution
Unidirectional		
a followed-by b	a	$a; b$
a preceded-by b	a	$b; a$
a replaced-by b	a	b
Bidirectional		
a merge b	a	$a; b$
= a followed-by $b \wedge b$ followed-by a	b	$b; a$
a swap b	a	b
= a replaced-by $b \wedge b$ replaced-by a	b	a

Table 2. Method correspondence operators

Table 2 shows the correspondence operators and their meaning in terms of method calls and executions. The semicolon is used to denote sequence: $a; b$ means that method a will be executed and immediately followed by the execution of method b.

The bidirectional operators merely combine unidirectional ones and exist to add brevity to specifications. Many others are possible besides the two shown previously. Figure 3 shows the syntax of the composition language.

$context$ ::= "{" $\{expression\}$ "}"
$expression$ ::= $name[context]$ | $name[context]$ $operator$ $name[context]$
$operator$ ::= followed-by | preceded-by | replaced-by | merge | swap

Fig. 3. Composition language syntax

Composition expression examples:

$a\{b$ **followed-by** $c\}$ Calls to a will result in the execution of a. During the execution of a, calls to b will result in the execution of b, followed by the execution of c.

$x\{p$ **followed-by** $q\}$ **preceded-by** $y\{r$ **swap** $q\}$ A call to x will result in the execution of y followed by the execution of x. In the execution of y, calls to r are replaced with the execution of q, and calls to q are replaced with the execution of r. In the execution of x, calls to p will result in the execution of p followed by the execution of q.

4.3 Object Binding

The composition operators are described at the class level. However, their effects are at the object level. A correspondence operator defines what happens when a call is made to a specific object. That call may result in the execution of methods in a different object. We need a way to determine the object referred to in the execution. Once that is determined, objects are bound to each other throughout their lifetime.

If there is a correspondence expression that matches a method call in a class a to a method execution in a different class b, there must be a binding expression detailing

how objects of class a are to be bound to objects of class b. A binding expression has the form *a operator b*, where a is the class that has the method called, b is the class that has the method executed in response to the call, and $operator$ is a binding operator. There can only be one binding expression involving any given pair of classes.

Currently, our language supports only three binding operators: *binds-to-unique*, *binds-to-any*, and *binds-to-all*. The expression *a binds-to-unique b* means that an object of class a is bound to any object of class b, as long as b has not yet been bound to any other object of class a. If such an object does not exist, one must be created. Another kind of binding is *binds-to-any*. The expression *a binds-to-any b* means that an object of class a can be bound to any existing object of class b. Finally, *binds-to-all* means that an object of class a will be bound to all existing objects of class b.

The effect of *binds-to-unique* is to create a one-to-one correspondence between objects. If *a binds-to-unique b*, then for each object of class a there will be an object of class b. The effect of *binds-to-any* is to create a many-to-one correspondence. If *a binds-to-any b*, a single object of class b is sufficient; all objects of class a can bind to that object. Finally, *binds-to-all* creates a many-to-many correspondence. In this case, when a method is called on an object of class a, the corresponding method of class b will be executed for all objects of class b.

5 Example: Concurrent File System

In this example we will take two independent modules, a simple file system and a concurrency control unit, and consider them as two hyperslices, or separate aspects, of an integrated system. The system is to give support for concurrency to the file system.

Hyperslice 1: file system A simple file system hyperslice contains two classes: `File` and `Directory`. Both allow *read* and *write* operations.

Hyperslice 2: shared buffer The shared buffer hyperslice contains a single class, *Mutex*, which encapsulates a shared buffer for use in a concurrent environment by multiple threads. The shared buffer contains three methods: *read*, *write*, and *cs*. Many readers can access the buffer at the same time, but writers require exclusive access. The *cs* method implements the critical section, which is where the buffer is actually manipulated.

Our intention is to combine the two hyperslices to produce a file system that supports execution in a concurrent environment, i.e., that allows many simultaneous users to read from a file or directory, but requires exclusive access to write in either of them. Figure 4 shows the class definitions for the two hyperslices.

```
class File { read( ), write( ) }
class Directory { read( ), write( ) }
class Mutex { read( □ cs ), write( □ cs ), cs( ) }
```

Fig. 4. Class definitions

The *Mutex* class provides synchronization for the desired system. Our intent is for users to still be able to use the File and Directory classes normally. However, before the file system operations can be used, the synchronization process must happen. This means that the appropriate Mutex method must be executed before the file system method. We use a correspondence expression to make sure that this happens, and that the appropriate file system method is executed when the Mutex object enters the critical section. The four required correspondence expressions are shown in figure 5.

```
File.read replaced-by Mutex.read { cs replaced-by File.read }
File.write replaced-by Mutex.write { cs replaced-by File.write }
Directory.read replaced-by Mutex.read
            { cs replaced-by Directory.read }
Directory.write replaced-by Mutex.write
            { cs replaced-by Directory.write }
```

Fig. 5. Correspondence expressions

The four expressions have similar structures. Let's examine the first line. When the user calls the read method of a File object, the read method of a Mutex object will be executed instead. The method will go through the read access protocol for the shared buffer. When it becomes possible to read the shared buffer, the cs method will be called. However, the context specified that File.read be executed instead. The file is then read. When File.read returns, the rest of the Mutex.read method is executed, releasing any locks that may be necessary. Figure 6 shows the flow of control between the hyperslices.

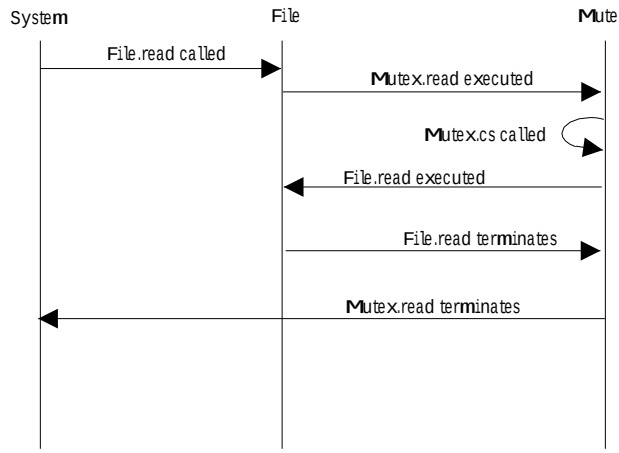

Fig. 6. Flow of control after correspondence

```
File binds-to-unique Mutex
Directory binds-to-unique Mutex
```

Fig. 7. Object binding expressions

The effect of the binding expressions, shown in figure 7 is to ensure that each *File* or *Directory* object will have its own *Mutex* object. This will allow each file to have its own concurrency access; while a file is being read, a different file can be written. Had we used *binds-to-any*, instead, we would have all files sharing a single Mutex object, which would have the effect of only allowing the file system to write to a single file at a time. This illustrates how the behavior of the system can be modified by changing the binding expressions.

6 Work in Progress

Many issues still need to be tackled before the model can be used as a basis for the extension of object-oriented approaches. The language is still untyped, and methods support neither arguments nor return values. Dealing with these will require work on the semantics of the correspondence operators, since the parameters or return types of a called method may differ from those of the executed method. The language also needs support for inheritance and more complex types of object binding.

We use transformations to achieve the desired semantics mandated by the correspondence operations. Since conventional object-oriented languages offer no primitives that are equivalent to these operators, the model becomes more useful if the systems described can be converted into systems in the conventional object-oriented model. This can be accomplished by a set of semantics-preserving transformations that would modify the existing classes to apply the functionality dictated by the correspondence operators.

The formal model underlying the language is under development using the PVS language and theorem prover [17].

7 Conclusion

We are providing the formal infrastructure for reasoning about multiple-perspective object-oriented systems. Our model is based on method-level correspondence, and is simple yet capable of describing the complex relationships that are necessary for this form of separation of concerns. We use call graphs to define the structure of individual hyperslices. Correspondence operations are used to specify changes in the call graph structure, and binding operators specify how runtime objects are matched. The example shown in this work is small, but illustrates the potential for using the model languages in hyperslice composition.

Once the model has been further expanded to include more object-oriented mechanisms such as inheritance, we intend to use it for formal reasoning about systems defined in many of the existing approaches such as subject-oriented programming. In particular, we are interested in consistency checking, a constant problem when using

this form of decomposition. We also wish to use our semantic infrastructure as the basis for extending other object-oriented languages with multiple-perspective capabilities.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Paulo Alencar, Donald Cowan, Carlos Lucena, and Torsten Nelson. Viewpoints as an evolutionary approach to software system maintenance. In *Proceedings of IEEE International Conference on Software Maintenance*, October 1997.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[4] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. Nilson, and W. Schnure. A feature interaction benchmark for IN and Beyond. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 1–23. IOS Press, 1994.

[5] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1989.

[6] Steve Easterbrook and Bashar Nuseibeh. Using ViewPoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43, January 1996.

[7] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

[8] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, 1979.

[9] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA '93*, pages 411–428. ACM, 1993.

[10] I. M. Holland. Specifying reusable components using contracts. In *Proceedings of ECOOP '92*, 1992. Lecture Notes in Computer Science no. 615.

[11] ISO. *ITU Recommendation X.901-904 – ISO/IEC 10746 1-4: Open Distributed Processing – Reference Model – Parts 1-4*. ISO, 1995.

[12] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[13] Michael Jackson. *Software Requirements and Specifications: a lexicon of principles, practices and prejudices*. Addison-Wesley, 1995.

[14] Gregor Kiczales, J. Lamping, A. Mendhekar, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP '97*, 1997.

[15] Bent Bruun Kristensen. Roles: Conceptual abstraction theory and practical language issues. *Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity on Object-Oriented Systems*, 1996.

[16] Harold Ossher and Peri Tarr. Operation-level composition: a case in (join) point. In *Proceedings of the 1998 ECOOP Workshop on Aspect-Oriented Programming*, 1998.

[17] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[18] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[19] Mark Skipper and Sophia Drossopoulou. Formalising composition-oriented programming. In *Proceedings of the 1999 ECOOP Workshop on Aspect-Oriented Programming*, 1999.

[20] Peri Tarr, Harold Ossher, William Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.