Towards a formal model of object-oriented hyperslices

Torsten Nelson, Donald Cowan, Paulo Alencar

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

Abstract

This position paper presents work in progress on a formal model for the composition of object-oriented hyperslices with method-level join points. With the formal 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

Multi-Dimensional Separation of Concerns (MDSC) is a model of decomposition that seeks to remedy the deficiencies of traditional methods of decomposing problems [Tarr et al., 1999]. A problem decomposed according to MDSC is formed by a set of hyperslices, where a hyperslice contains all elements that address a specific concern of the system. MDSC can be applied at any level of abstraction.

MDSC distinguishes itself from other approaches to decomposition by the fact that the parts that make up the decomposed problem are not disjoint. In other approaches, 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 hyperslices, and its definition can be different in each hyperslice.

The hyperslice concept is directly related, and similar to, the concepts of “views” and “viewpoints” in software engineering. These terms have become quite overloaded with similar but slightly different meanings. The exact definition depends on different authors and different methods. Daniel Jackson, for example, offers the following definition: "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" [Jackson, 1995]. We favor this definition of "view" with regard to hyperslices - a hyperslice specifies some aspect of the entire program.

1.1. Object-oriented hyperslices

While the concepts behind MDSC 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.

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 class. 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 invoke the output, 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.

1.2. 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 objects, however, that exist in various hyperslices. In order to form the desired system, we must establish the *correspondence* between these objects.

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 behaviour of the system. Matched classes may have complementing behaviour, or one's behaviour 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*) 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. In [OT99], Ossher and Tarr argue in favor of this approach. 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.

2. 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 formally specify 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.

Once completed, the model should allow us to study properties of techniques that use operation-level correspondence such as subject-oriented programming [Harrison and Ossher, 1993], as well as serve 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.

2.1. Class language

The class language defines classes as sets 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 yet 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 [Abadi and Cardelli, 1996].

Syntax:

hyperslice ::= *class* { *class* }
class ::= **class** *name* **{** *method* {**,** *method* } **}**
method ::= *name* **(** { *called-method* } **)**
called-method ::= □ *name* | ◊ *name*
name ::= an identifier containing letters, hyphens, or a period. The period separates class names from method names.

Example:

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

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`**).

2.2. 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.

Table 1 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*.

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* followed-by *b* ∧ *b* followed-by *a*	*a* *b*	*a; b* *b; a*
a swap *b* = *a* replaced-by *b* ∧ *b* replaced-by *a*	*a* *b*	*b* *a*

Table 1: Method correspondence operators

The bidirectional operators merely combine unidirectional ones and exist to add brevity to specifications. Many others are possible besides the two shown above.

Composition language syntax:

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

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*.

2.3. Object binding

The composition operators are described at the class level. However, its effects are at the object level. A correspondence operator says 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 exactly what object the execution refers to. 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.

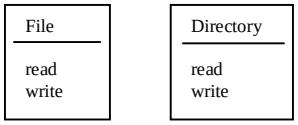
Currently, our language supports only three binding operators: *binds-to-unique*, *binds-to-any*, and *binds-to-all*. The expression *a binds-to-unique b* means that an object of class *a* is bound to any object of class *b*, as long as *b* has not yet been bound to any other object of class *a*. If 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*.

3. 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.

Class definitions:

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

The Mutex class is the synchronization aspect 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 below:

Correspondence:

```
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 }
```

The four 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 1 shows the flow of control between the hyperslices.

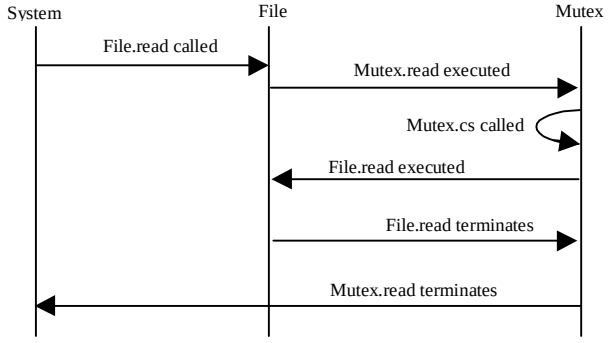


Figure 1: Call graph after correspondence

Binding:

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

The effect of the binding expressions 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 to. 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.

4. 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 substantial 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.

Another important issue under development is the use of transformations. The model describes the effect of correspondence operators over a set of classes. However, since conventional object-oriented languages offer no equivalent to these operators, the model would be more useful if the systems described in accordance to it could 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.

5. Related work

The *Ku* language [Skipper and Drossopoulou, 1999] is a formal model for hyperslices that follows much the same vein as the one presented here. *Ku* has different goals, however, meaning to study a broader range of composition technologies, including aspect-oriented programming. It allows class definitions with typed attributes, and method bodies with assignment. The composition operators in *Ku* operate at the class level. Our approach deals solely with the effect of composition on method calls. We feel that our work is complementary in nature to that of *Ku*, being directed to a narrower domain.

References

[Abadi and Cardelli, 1996] M. Abadi and L. Cardelli, *A Theory of Objects*. Springer-Verlag, 1996.

[Harrison and Ossher, 1993] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA 1993*, pp. 411-428. ACM, 1993.

[Jackson, 1995] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*. Vol. 4, no. 4. October 1995, pp. 365-389.

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

[Skipper and Drossopoulou, 1999] M. Skipper and S. Drossopoulou, Formalising Composition-Oriented Programming. In *Proceedings of the 1999 ECOOP Workshop on Aspect-Oriented Programming*.

[Tarr *et al*, 1999] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr., *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*. Los Angeles, May 1999.