ASPECT-ORIENTED PROGRAMMING

GREGOR KICZALES, JOHN IRWIN, JOHN LAMPING, JEAN-MARC LOINGTIER, CRISTINA VIDEIRA LOPES, CHRIS MAEDA, ANURAG MENDHEKAR

XEROX PALO ALTO RESEARCH CENTER

1. INTRODUCTION

In this paper, we present an overview of our recent research on programming language expressivity. The goal of this work is to make it possible for programs to clearly capture all of the important aspects of a system's behavior, including not only its functionality, but also issues such as its failure handling strategy, its communication strategy, its coordination strategy, its memory reference locality, etc.

Our current work is based on the belief that programming languages based on any SINGLE abstraction framework—procedures, constraints, whatever—are ultimately inadequate for many complex systems. The reason is that the different aspects of a system's behavior that must be programmed, each tend to have their own "natural form", so while one abstraction framework might do a good job of capturing one aspect, it will do a less good job capturing others.

This conclusion has led us to develop a concept we call Aspect-Oriented Programming (AOP). In AOP, the different aspects of a system's behavior are each programmed in their most natural form, and then these separate programs are woven together to produce executable code. Our work on AOP is being carried out in the context of both general-purpose and domain-specific languages, we believe that it has contributions to make to both areas.

1.1 CROSS-CUTTING

The basic limitation of single abstraction framework languages is that one abstraction will not necessarily serve well for all the different issues that must be programmed in a specific system. A classic example is the notion of invariant relations among objects. While many standard object-oriented languages do a good job of clearly capturing the behavior of objects, they do a less good job of capturing structural and behavioral invariants, such as "when this object gets a `pop` message, send this other object a `refresh` message." Many linguistic mechanisms have been developed to deal with special cases of this problem (i.e. before/after methods), but a great deal of the complexity in real world code still appears to come from cases where the language fails to provide adequate support for a secondary, but still important, aspect of a system's behavior.

Some cases of this problem can be dealt with by isolating one issue in one part of the code, and another issue in another part of the code. This is often done using some form of procedure abstraction. For example, the details of memory allocation can be hidden behind the `malloc/free` interface, the client code only attends to what to do with the objects.

But there are some cases where two aspects of a system's behavior seem to invariably get tangled together in the code. In such cases we say the two aspects "cross-cut" each other with respect to the program. One good example happens in object-based distributed computing. There have been several efforts to isolate the work of deciding how much of an object to send along in remote message sends (marshalling). But analysis of real programs shows that it nonetheless tends to get mixed in throughout the code. The reason is that deciding the optimal marshaling strategy requires knowing a lot about each particular send, and so a lot of the code tends to gravitate towards the actual sends and out of the sub-module.

We believe this TANGLING-OF-ASPECTS phenomenon is at the heart of much of the complexity in existing software systems. Further, we believe that increasing the level of programming languages won't help without addressing this root cause of the tangling. Instead what is needed is to be able to work with abstractions that correspond more directly to all the different aspects of concern in a system.

The goal of Aspect-Oriented Programming (AOP) is to make it possible to deal with cross-cutting aspects of a system's behavior as separately as possible. We want to allow programmers to first express each of a system's aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver™. The name weaving is chosen to reflect an important difference from traditional compilation—in weaving, the output is a much tighter integration of the input programs than in traditional compilation. This follows directly from AOP's goal of separating traditionally cross-cutting issues.

Examples of the kinds of aspects we believe AOP should allow programmers to think and program in terms of are shown in Table 1. Note that programming in terms of aspectual decomposition requires much more than just identifying the different aspects of concern. It requires being able to express those aspects of concern in a way that is precise and that makes the RELATIONS among the aspects of concern precise. This is what enables the Aspect Weaver to work, and is also what enables reasoning about the code, debugging the code, and all other parts of the program lifecycle.

Domain:	image processing	distributed computing	operating systems
Aspects:	operations on pixel maps	what the objects do	algorithm
	control structure	their location	data structure
	intermediate value sharing	communication	reference locality
		synchronization	

Table 1- Some example domains and key aspects of concerns in each.

We believe that a number of projects, new and old, include intuitions similar to those underlying Aspect-Oriented Programming. The contribution of this short position paper are to: (i) Provide an initial sketch of the EXPLICIT notion of AOP; (ii) Present the first systems to be constructed with AOP an as an explicit guiding principle; (iii) Discuss some of the problems that

arise when attempting to provide separate control over cross-cutting aspects of a system's behavior.

2. AN EXAMPLE OF ASPECT-ORIENTED PROGRAMMING

As a first example of AOP, we present a small part of one of the projects we have been working on—using AOP to handle a representative class of distributed applications. The aspectual decomposition we are working with in this domain breaks systems down into several key aspects, including: the basic functionality of objects, the communication strategy when messages are sent across address space boundaries, and coordination of the threads of activity

We have developed a basic functionality language (BFL) and appropriate aspect description languages (ADLs) to capture each of these different aspects of the system's behavior. The BFL is a simplified C++/Java style language. Programs in this language specify what the objects ***do,*** in the familiar style of imperative object-oriented languages. Programs in the communication ADL can specify what parameters should be copied, and to what extent, when there are method invocations between objects in different address spaces. Programs in the coordination ADL define sets of methods that are mutually exclusive and/or auto-exclusive; and define pre-conditions on the execution of methods. The code below is the three programs that define a bounded stack in our system. (Note that all the figures in this paper were originally produced in color, which helps to capture the aspect weaving more clearly. A color version of the paper is available at http://www.parc.xerox.com/aop.)

Basic Functionality

```
class BStack {
  Integer head;
  Element elts[MAX];
  …

  void! insert (Element e) {
    if (head == MAX) !;
    else elts[head++] = e;
  }
  Element! remove ( ) {
    if (head == 0) !;
    else elts[--head];
  }
  Element! top ( ) {
    if (head == 0) !;
    else elts[head];
  }
  void newClient (Client c){
    …
  }
};
```

Communication Aspect

```
interface BStack {
  void! insert (gref Element);
  gref Element! remove ( );
  gref Element! top ( );
  void newClient (copy Client:id);
};
```

Coordination Aspect

```
relax BStack {
 autoex{insert, remove, new-
Client};
 mutex {insert, remove};
 mutex {remove, top};
}
```

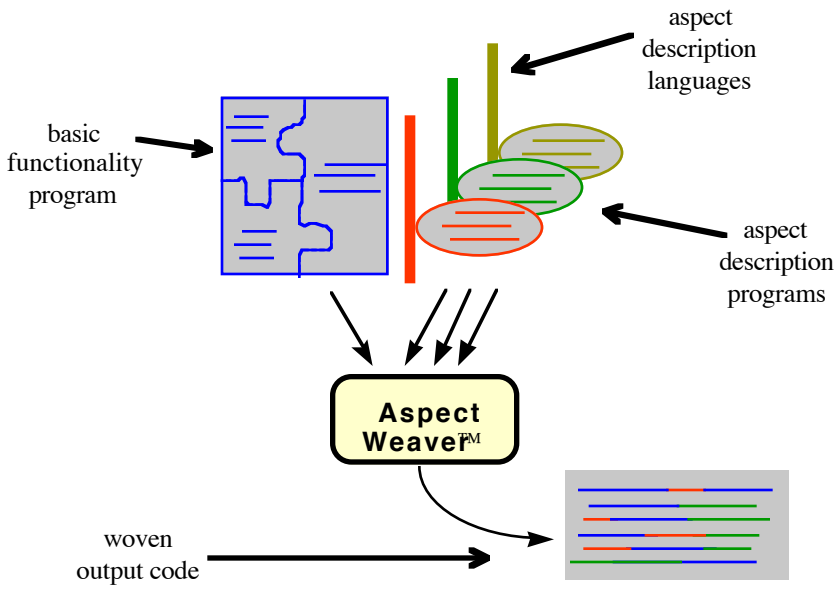


Figure 1—The basic elements of an aspect-oriented programming system. The basic functionality (or primary aspect) is captured using a language that best suits it. Each of the cross-cutting aspects are captured using other appropriately specialized languages. The weaver takes all the programs as input and produces woven output code, which may itself be source code in a language like C.

Before these programs can be run, they must be woven together to produce executable code. This is done using a tool we call an Aspect Weaver, as shown in Figure 1. In all of our current implementations the Aspect Weaver actually outputs a medium-level language like C, C++ or Java. That code is then compiled using a traditional compiler. (The woven code for this example is listed in the appendix.)

This example shows the power of the aspectual decomposition and AOP. The aspectually decomposed program is easy to write, maintain and reason about. Design and implementation of the weaver is moderately straightforward. The output of the weaver, that is the automatically tangled code, is quite efficient.

In this example, the aspect weaving is tightly connected to object invocations. This makes the weaver relatively straightforward to design and implement. Our second example involves more complex weaving, and so provides the basis for discussing some of the technical issues we see in developing AOP into a working programming paradigm.

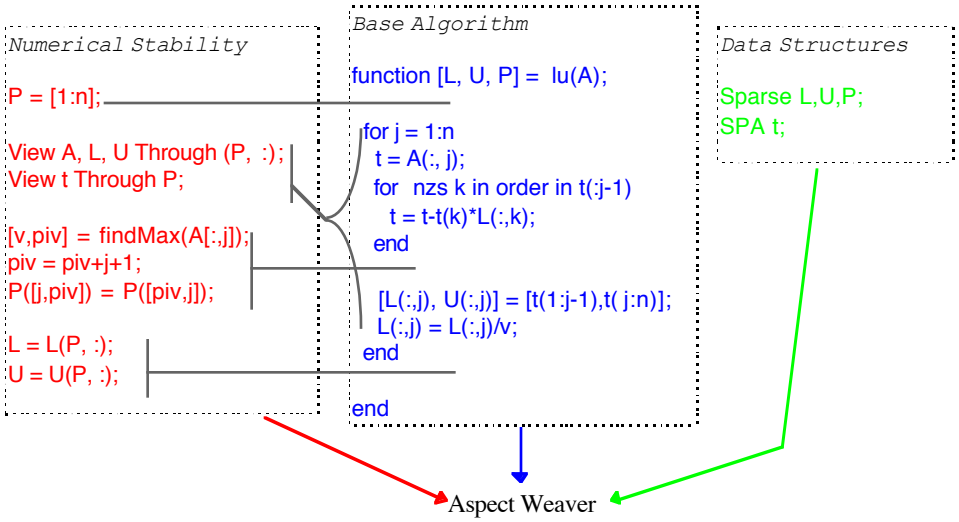
3. A SECOND EXAMPLE OF ASPECT-ORIENTED PROGRAMMING

Our second example of aspect-oriented programming is from the domain of scientific computing, where we have been exploring the use of AOP in solving differential equations, cast as solving sparse matrix equations.

The decomposition we have chosen for this problem has three main aspects: (i) the basic algorithm to be used to solve the system (often a custom algorithm tuned to the physics of problem at hand); (ii) maintaining numerical accuracy (which can require monitoring the computa-

tion and permuting the original matrix on the fly without losing track of the original relationships); (iii) choosing data structures that have the appropriate space/time tradeoffs and allow sharing of information across iterations of the algorithm.

As in the previous example, to work with this decomposition, we use three languages that together make it possible to express the three aspects of the desired computation in a natural and coordinated way. The figure below illustrates how these languages work, by showing how LU factorization is coded. Again, this figure shows what the programmer writes. The code in blue—the algorithm aspect—is just the simple, elegant, algorithm that appears in textbooks. The language for this aspect is a simple variation of Matlab. The other aspects are equally clear to an engineer versed in this domain. The gray lines in the figure are a kind of hypertext mechanism that connects the descriptions of the aspects. The bottom of the figure shows the tangled code that results from weaving the aspects together. The color in the bottom shows the actual weaving of the three separate aspects.




```
Void lu(Sparse Matrix &A, Sparse Matrix &L, Sparse Matrix &U, Permutation &P) {
 int n = A.size(); L.set.size(n,n); U.set_size(n,n);
 P.initialize(n);
 SPA t;
 for (j=1; j<=n; j++) {
  Sparse Vector c = A.column(j, P);
  load(t, NoRange, NoHeap, c, NoRange);
  Enumerator e(t, P , Range(1, j-1));
  for ( k=e.get_top(); k!=0; k=e.get_top())
   dpaxpy(t,P,NoRange,Heap,A.column(k,P),-t.get_value(k,P,NoRange));
  int piv; Value v;
  findMax(t,P,Range(j,n),piv,v);
  P.swap(j,piv);
  split(t, P,NoRange,L.column(j,P),NoRange ,U.column(j,P),NoRange ,j);
  L.column(j,P).divide(P,NoRange ,v);
 }
 L.permute(P);
 U.permute(P);
}
```

Figure 2—Three different aspect description programs and their resultant weaving in AML.

All of this would only be a nice story if it wasn't for the quality of the output code. It is just as fast as the corresponding tangled code from the standard Fortran library, and is 100 times faster than running the code for the basic algorithm on the newest commercially-available compiled MatLab system.

This example, again shows the power of the aspectual decomposition idea. This system is easy to program in, maintain, reason about, and is remarkably efficient. The numerical analyst we developed it for says it is "as easy to think about as Matlab code, and as fast as well-tuned Fortran code." As a specific example, he says "[when working in the tangled way] I always used to make mistakes about where to put in the pivot permutations, but now I don't."

This power stems from the fact that each of the important aspects of concern to the programmer is being addressed separately, and in a first class way. This contrasts with both the traditional tangled approach, where all aspects of concern are addressed by the programmer, but in a tangled and confused way, and with the traditional high-level approach (i.e. Matlab), where only the algorithm aspect gets addressed by the programmer, and the compiler is left to divine the other aspects.

4. SEPARATION WITH COORDINATION

These examples serve to clarify an earlier point about AOP—Aspect Weavers don't separately compile the different ADPs and then have them call each other across procedure call interfaces. Remember that the goal of AOP is to make it possible to separately express different cross-cutting aspects of a system's behavior. This is why the relation between the input ADPs and the output of the weaving looks the way it does in Figure 2.

This means that the ADLs must be designed so that the different ADPs are not completely separate, but are instead separate but coordinated perspectives on the total computation. This section discusses issues that flow from this requirment, which we see as the most fundamental technical issues in developing the AOP paradigm.

Looking carefully at the sparse matrix example, we see that the basic algorithm aspect sees the computation much as an ordinary program does—an operation, like `t=A[:,j]`, in this aspect will be done once each time control passes through it. The declarations of the data structure view, on the other hand, are associated with a data object—a declaration, like `Sparse A`, specifies that the matrix named `A` should be represented with a sparse data structure, wherever it might go, even if it is passed to other procedures. A single declaration in the data structure view can thus have an effect on multiple executions specified throughout the basic algorithm aspect.

This is similar to the way in which the communication aspect of the distributed queue application, says "don't copy or move objects that queues see." This applies to all the operations that operate on queues and objects, for all queues.

In each case, the different aspects coordinate with each other and with the overall computation differently. A key focus of our work is to master this range of ways that aspects can interrelate, both in designing ADLs and in building Aspect Weavers. To approach this problem, we are working with two main themes: exploiting the concept of base/meta separation, which provides an initial intuitive approach for designing these coordinations; and developing a solid con-

ceptual analysis of the correspondence relations to provide a more long-term and formalizable foundation.

4.1 EXTENDING BASE/META SEPARATION

A simple approach to achieving separation and coordination comes from extending the ideas of base/meta separation found in metaobject protocols and reflection. The original idea behind base/meta separation is that a system can have two interfaces: one which provides the basic functionality, and one which can be used to ask about, monitor or adjust the functionality available through the first interface.

Base/meta separation can be extended into AOP by designing a base aspect to capture "what to do" and other aspects to capture various aspects of "how to do it." By having all the auxiliary aspects relate to the base one, their coordination is simplified, and it becomes easier to design an Aspect Weaver that can put them all together.

In the distributed computing example, we used C++ as the base aspect description language. The other aspects address how the base aspect is implemented. In the sparse matrix example, we used a simple variant of the Matlab language as the base aspect description language. The numerical stability aspect (red) controlled the iteration order through arrays, while the data structures aspect (green) controlled the representation of the arrays and of iterations over them.

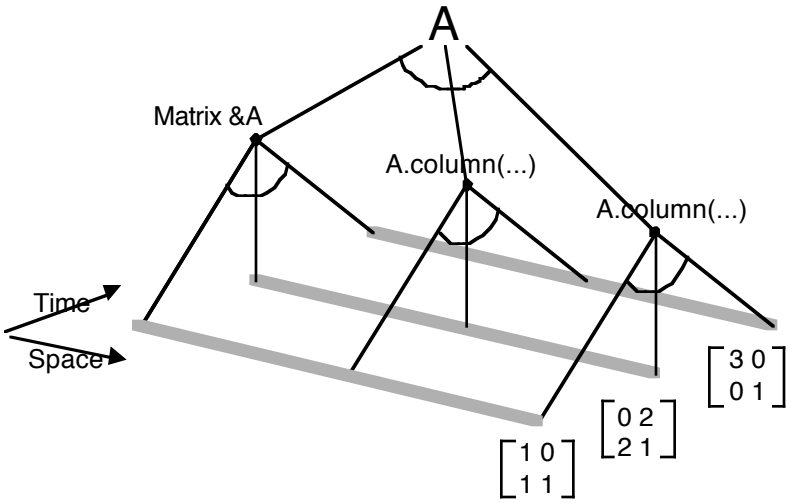
4.2 ANALYSIS OF CORRESPONDENCE

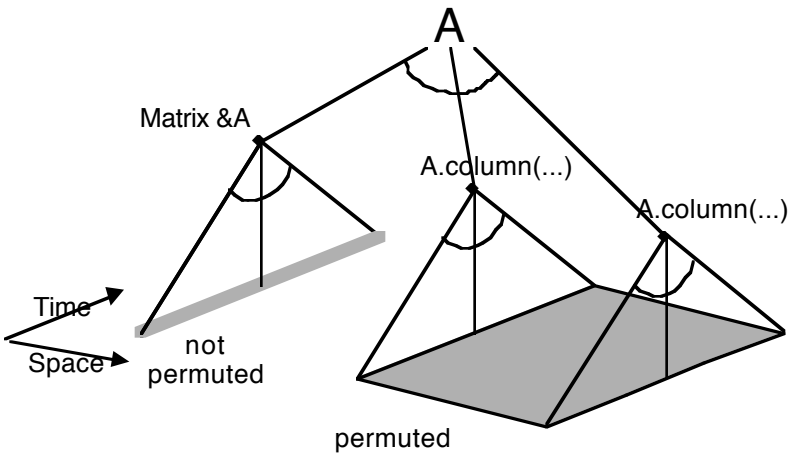
While the base/meta separation principle has proven to be quite powerful, it has three weaknesses that may make it insufficient as a complete foundation for AOP: (i) it is informal, (ii) it makes one aspect primary over the others, and (iii) it relates the aspects to each other, rather than relating them to the complete computation.

Our goal is to develop an explicit analysis of the correspondence relations in AOP, that gives us a a clear and formalizable understanding of the ways in which different aspects coordinate with each other and with the final computation. We believe that it will be possible to formalize this analysis into a "correspondence calculus" that could play a role similar to that which the lambda calculus has played in analyzing and building systems based on procedure abstractions.

We have developed some initial ideas for this analysis, based on the observation that the structure of connections between the aspects and the executable code is largely one of fan-outs and fan-ins, and that it is the interaction between these that leads to the way in which different aspects cross-cut each other. As an illustration of these ideas, consider the variable `A` in the output code of the fluid dynamics example, and the matrix that it indicates.

First consider the situation from the point of view of the basic algorithm aspect, as shown to the right. There are several occurrences of `A`, fanning out in the spatial dimension. Each of these will be encountered several times during execution, fanning out in the temporal dimension. But, seen from this aspect, `A` refers to the same matrix throughout a particular invocation of `lu`. The gray bars at the bottom, reflect this, they show the fan-in onto individual matrices.





Now consider the situation from the point of view of the numerical stability aspect. The first two levels of fan-out are the same. But seen from this aspect, the first occurrence of `A`, in the procedure argument, refers to a non-permuted view of the matrix, while all the other occurrences refer to a permuted view. This is reflected by the wide gray bar, indicating a single region of fan in.

What these pictures help capture is the way in which the language for a particular aspect fans out from what is syntactically local to that aspect (i.e. a data structure declaration) to a multitude of execution activities (i.e. every activity involving that data structure), which fan back in to coherent concepts of the aspect (i.e. the data structure, itself).

These pictures also give us a clear view of the way in which the two ADPs provide a precise handle on cross cutting issues. The execution activities form a common ground plane across which the aspectual decompositions cross cut, where all the pictures can be superimposed. In the illustration, one view identifies ``lines along space'' and the other “lines along time.”

5. NO SMART COMPILERS

A long-standing obstacle to making high-level programming languages work in performance critical domains is the difficulty of developing sufficiently efficient compilers. This problem has been an obstacle to both general-purpose and domain specific high-level languages.

Our approach to this problem is similar in spirit to the very idea of domain-specific languages. Whenever we are confronted with a hard implementation strategy problem for a BFL we put that problem under the control of a special ADL. This solution is similar in spirit to the idea of domain-specific languages itself.

Consider for example, the distributed computing example above. The traditional approach to simplifying application programs is to design a distributed object language that hides distribution issues (aka makes them transparent). But this is extremely difficult to do, since deciding how much of the objects to copy for remote message sends requires knowing a lot about the application's behavior. The traditional approach to making application programs efficient is for the programmer to circumvent the object model, and tangle the communication strategy in with the basic functionality.

We do neither of these. We don't try to virtualize hard implementation strategy problems, but we also don't let them get tangled in with the application's basic functionality. By separating the control over problems like communication strategy, we let the programmer use their knowledge of the application to control them, while ensuring that the code that does so will be clear and easy to maintain.

Asking the programmer to explicitly address implementation aspects may sounds like a step backwards, but our experience with open implementation suggests that it is in fact an important step forwards [2, 3, 4, 5, 6]. Programmers can detail with implementation strategy quite well. What they can't deal with as well are implementation details. So the key point is to ensure that the ADLs are well enough designed that programmers are focusing on implementation strategy, not implementation details. The control over implementation strategy must be at an appropriately abstract level, through an appropriate aspectual view, with appropriate locality.

6. RELATED WORK

A great deal of work, new and old, appears to be based on intuitions similar to those underlying Aspect-Oriented Programming. That is, the work decomposes systems along lines that feel more aspect-based than module-based. These examples, facilitate the study the issues of correspondence among aspects which are so important to developing the general paradigm.

The literature associated with some of these examples makes an explicit point of using a different kind of decomposition; for others it does not. But all of them have important similarities to AOP in that they are based on similar intuitions. They differ from AOP in the degree to which they (i) focus on the GENERAL paradigm, and (ii) support AUTOMATIC weaving.

6.1 METHODOLOGIES

Many design disciplines are based on well-established aspectual decompositions. For example, mechanical engineers use static, dynamic and thermal models of a system as part of designing it. Electrical engineers use various diagrams to reason about their circuits: circuit diagrams, waveform diagrams (timing diagrams), phase diagrams. Each of these models help isolate certain aspects of the circuit. [7]. All of these are aspect-based decompositions, but in these cases the weaving together aspects is mostly done manually, although certain CAD tools do some of it automatically.

Software engineers also do this, although often only informally. It is common to hear programmers say something like: "The real issues in this system are X, Y and Z. This line of code is written this way because I need to be sure that BOTH X and Y are satisfied." Some software development tools explicitly support certain aspectual decompositions. For example tools for

OMT methods let programmers draw different pictures of how objects should work. The notion of traceability in some design methodologies is in large part intended to help with the complexity that arises when manually weaving cross-cutting aspects of a system's behavior into tangled executable code.

6.2 ANALYTIC TOOLS

Work on Program Slicing is similar to AOP in that it recognizes the way that a given segment of program text may actually be a cross-cutting tangle of different issues. Program slicing makes it possible to tease out some of these different aspects of a program's functionality. Work on program slicing differs from AOP in two important ways: (i) it works with programs written entirely in the same language, and (ii) it is analytic rather than constructive in nature. One result of this is that the slices program slicing tools make available tend to be defined in terms of the common language, i.e. accesses to a given variable, callers of a specific function, dependents of a given module etc. We are interested in using program slicing techniques as a basis for Aspect-Oriented analysis and re-engineering tools.

6.3 CONSTRUCTIVE TOOLS

There are other systems, however, that are more like our concept of AOP in that the aspectual decomposition is constructive rather than analytical. Much of the work in the area of reflection can be viewed in this way. In this work, there is partial aspectual decomposition between "base" code, which looks like a program written in a traditional high-level language and "meta" code, which affects how the base code is implemented. Whereas the subject matter of the base code has to do with the specific domain of the program (i.e. paychecks, scanned document images), the subject matter of the meta-code has to do with the semantics and implementation of the language the base code is written in. The effect of the meta-code inherently cross-cuts the base-code—the two kinds of code deal with a different cut on the overall system's behavior. This is what give reflective techniques their appeal. For example, meta-code can have easy access to every message send, or every message send across machine boundaries. The base code on the other hand only has easy access to particular message sends (actually only to the initiation and receipt of the message, not its actual transmission). Different researchers have explored a wide variety of different metaobject protocol architectures to provide different relationships between the base- and meta-levels [8,9,10].

Some object-oriented analysis and design methodologies have automated support tools that can be seen as a specific cases of aspect weaving (see [11], for example). They deal with aspects of object-oriented program structure, such as: class hierarchy, object interaction, timing of message interaction and program modularization. The tools automatically generate programs from aspectually decomposed specifications. They don't however, support a general notion of aspectual decomposition.

Adaptive Programming (AP) is another example of a specific instance of AOP [12, 13]. AP is focused on separation between algorithm and data structure, but it allows greater separation than traditional ADT techniques. The operations are written in a highly data-structure neutral way, by having them access the object structure (class graph) using a kind of relational query language. This allows the same operation to be reused with different concrete data-structures. There is an explicit weavin step, in which the operations are tailored to a specific set of data structures, making sure there is no remaining runtime overhead from the strong separation.

Composition Filters (CF) [17] can also be seen as a specific case of AOP. In CF, the filters provide separate control over typical cross-cutting aspects such as communication, synchronization etc. Work on composition filters (like work on adaptive programming) has focused on the

OO domain. The filters work by intercepting messages that an object receives and "filter" these messages to allow for compositionally.

The Synthetix[18] project, led by Pu and others, can also be seen as an example of AOP. Synthetix improves operating system performance by specializing OS code based on application invariants that only become known at run-time. The effect of individual invariants very much cross-cut the specialized code. Knowing, for example, that a file is being opened read-only cuts through the entire file system structure. The weaving in Synthetix makes extensive use of partial evaluation technology, much like some of our weavers do.

In hardware design, various projects are attempting to simplify the design process using some form of aspectual decomposition. The Rapid Prototyping of Application Specific Signal Processors (RASSP) program includes many such projects [14]. One of the approaches that the program aims to promote is the separation of communication, computation and control [15] in digital signal processing systems. The Ptolemy project[16], has tried a similar aspectual decomposition for VHDL, where they separate the specification of functional and behavioral issues[16]. Another such project, Graphical Rapid Prototyping Environment (GRAPE II) for rapid prototyping of signal processing and communications systems, uses an aspectual decomposition where the aspects are functionality, partitioning, scheduling and network topology. GRAPE provides an environment in which to combine these aspects into a running DSP program.

7. SUMMARY AND FUTURE DIRECTIONS

We believe that the concept of Aspect-Oriented Programming can be of significant value to programming language research and development projects. Coming to understand the reasons why some aspects of a system's behavior must cross-cut each other and the executable code has been a significant help to our efforts to develop general-purpose and domain-specific programming systems. Using the concept of AOP, our systems end up having several different domain-specific languages, one for each of the different aspects of concern that must be programmed.

We expect a number of exciting developments in AOP over the next few years. Some of the ones we are most interested in discussing with the POPL community include:

- Commercial application of AOP technology. What are the commercial advantages of using AOP technology? Can these advantages be quantified?

- Conceptual foundations for understanding cross-cutting and weaving. What can be said in general about how and why different aspects of a system's behavior cross-cut each other with respect to the executable code. What can be said in general about how weavers work? Can we develop a clear understanding of the different "natural shapes" that different issues have?

- Development of AOP toolkit technology. Can we develop reusable components to support building ADLs and weavers.

8. REFERENCES

1. Berlin, A., *et al.*, *Distributed Information Systems for MEMS*, *ISAT Study*, . 1995, ISAT.

2. Kiczales, G. *Towards a New Model of Abstraction in Software Engineering*. in *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*. 1992. Tokyo, Japan.

3. Kiczales, G., Lamping, J., Lopes, C., Mendhekar, A, Murphy, G, *Open Implementation Design Guidelines* in *Proceedings of the Internal Conference on Software Engineering* 1997. Boston, Massachusetts. (To appear.)

4. Kiczales, G., *Why are Black Boxes so Hard to Reuse?* 1994: Invited Talk, OOPSLA'94.

5. Kiczales, G., *Why Black Boxes are so Hard to Reuse*, 1995.

6. Kiczales, G., *Beyond the Black Box: Open Implementation*. IEEE Software, 1996. **13**(1): p. 8--11.

7. Fisler, K., *Exploiting the Potential of Diagrams in Guiding Hardware Reasoning*, in *Logical Reasoning with Diagrams*, G. Allwein and J. Barwise, Editors. 1996, Oxford University Press.

8. Yokote, Y. *The Apertos Reflective Operating System: The Concept and its Implementation*. in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*. 1992.

9. Okamura, H., Y. Ishikawa, and M. Tokoro, *AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework*, in *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*. 1992. p. 36-47.

10. Okamura, H. and Y. Ishikawa, *Object Location Control Using Meta-level Programming*, in *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, T.A. Pareschi, Editor. 1994, Springer-Verlag. p. 299-319.

11. Booch, G., *Object Oriented Design with Applications*. 1994: Benjamin/Cummings.

12. Lopes, C.V. *AP/S++: A Case Study of a MOP for Purposes of Software Evolution*. in *Proceedings of Reflection 96*. 1996. San Francisco.

13. Lieberherr, K.J., I. Silva-Lepe, and C. Xaio, *Adaptive Object-Oriented Programming Using Graph-Based Customization*. Communications of the ACM, 1994. **37**(5): p. 94-101.

14. Agency, A.R.P., *Rapid Prototyping of Application, Specific Signal Processing (RASSP)*. .

15. Harr, R.E., *Rapid Prototyping of Application, Specific Signal Processing (RASSP) Overview Presentation*, http://esto.sysplan.com/ESTO/RASSP/Presentation/index.html.

16. Buck, J.T., *et al.*, *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. Int. Journal of Computer Simulation, 1994: p. 155-182.

17. Aksit, M., *et al*. *Abstracting Object Interactions Using Composition Filters*. in *European Conference on Object-Oriented Programming, Workshop on Object-Based Distributed Programming*. 1993: Springer Verlag.

18. Pu, C., *et al*. *Optimistic Incremental Specialization: Streamlining a Commercial Operating System*. in *The 15th ACM Symposium on Operating Systems Principles*. 1995. Copper Mountain Resort, Colorado: ACM Press.

APPENDIX: WOVEN CODE FOR DISTRIBUTED SYSTEMS EXAMPLE

```
class QueueImpl {
  const int max = 100;
  int head;
public:
  void _ insert_impl(Element*);
  Element* _ remove_impl( );
  void _ insert_pack_in( pack_buf*,
Element*);
  void _ remove_pack_out( pack_buf*,
Element*);
  virtual void insert (Element *) { };
  virtual Element *remove( ) { };
};

class Queue : QueueImpl {
  QueueDup * mydup;
public:
  Queue( );
  void insert(Element *);
  Element *remove( );
};

class QueueDup : QueueImpl {
public:
  QueueDup ( );
  void insert(Element*);
  Element* remove( );
 };

Queue::Queue( ) {
  head = 0;
  mydup = remote_fork (" backuphost",
"QueueProg");
};

void Queue::insert(Element *e) {
  pthread_mutex_lock(&x);
  while (head == max)
    pthread_cond_wait(&_ insert_wait,
&x);
 this->_insert_impl(e);
  mydup->insert(e);
  pthread_cond_signal(&_ remove_wait);
  pthread_mutex_unlock(&x);
}

Element * Queue::remove ( ) {
  Element* retval;
  pthread_mutex_lock (&x);
  while (head == 0)
    pthread_cond_wait(& remove_wait,
&x);
  retval = this->_remove_impl( );
  mydup->remove ( );
  pthread_cond_signal(&_ insert_wait);
  pthread_mutex_unlock(&x);
  return retval;
}
```

```
void QueueImpl::_insert_impl(Element*
e) {
  if (head < max) elts[ head++] = e;
}
Element* Queue::_remove_impl( ) {
  if (head > 0) return elts[head—];
  return nil;
}
void
QueueImpl::_insert_pack_in( pack_buf
*buf,

Element *e) {
marshall( buf, e, " ref");
}

void QueueImpl::_remove_pack_out
(pack_buf * buf,

Element *e) {
marshall( buf, e, " ref");
}

void QueueDup::insert(Element *e) {
 this->_insert_impl(e);
}

Element * QueueDup::remove( ) {
 return this->_remove_impl( );
}


// Client code
Queue * aQueue = AnEmptyQ( );
void f( ) {
  Element e;
  for (;;) {
    aQueue->insert (e);
    aQueue->remove ( );
  }
}
void g( ) {
  for (;;) {
    Element *e = aQueue->remove( );
    aQueue->insert (e);
  }
}
int main ( ) {
  new_activity (f);
  new_activity (g);
}
```