Code Generation for WSLAs Using AXpect

Galen S. Swint and Calton Pu, *Senior Member, IEEE*

***Abstract*— WSLAs can be viewed as describing the service aspect of web services. By their nature, web services are distributed. Therefore, integrating support code into a web service application is potentially costly and error prone. Viewed from this AOP perspective, then, we present a method for integrating WSLAs into code generation using the AXpect weaver, the AOP technology for Infopipes. This helps to localize the code physically and therefore increase the eventual maintainability and enhance the reuse of the WSLA code. We then illustrate the weavers capability by using a WSLA document to codify constraints and metrics for a streaming image application that requires CPU resource monitoring.**

***Index Terms*— Software quality, Software tools, System software.**

I. Introduction

As the web services model gains popularity for developing and deploying applications, many companies are proposing standards and specifications to codify the constraints that necessarily must exist between parties involved in application. In fact, recent efforts of companies such as IBM and HP trend towards specifying web service contracts that capture expectations of performance and the roles of the involved parties [1][2]. We can view these specifications as domain specific declarative languages especially for the web services domain. The web service specification problem, then, becomes one of mapping the service contract's particular domain specific language (DSL) to the application's implementation space. At runtime, then, the measurement, contract evaluation, monitoring, and adaptive functionality must be interspersed into the application's normal runtime pattern.

Clearly, these standards specify an aspect of their associated web services. That is, they capture a system characteristic that is orthogonal to the primary application functionality, and that functionality crosscuts, or touches on many parts of, the application's implementation. Aspect-Oriented Programming (AOP) has recently emerged as a candidate development paradigm for managing code that implements application requirements that have the crosscutting characteristic. For general purpose languages, and by extension the general space of all applications, AspectJ and AspectC have shown that AOP techniques can significantly improve the clarity, and therefore maintainability, of application implementations. However, adding AOP to existing languages has also proven to be non-trivial. AspectJ is clearly successful, but it took several years of significant group effort. In comparison, progress on AspectC is slower. One of the major AOP research questions is whether this difficulty (of adding AOP) is related to the expressive power of the target application space or is simple inherent to AOP.

We investigate this question and probe the power of AOP in implementing these service contracts by using IBM's Web Service Level Agreement (WSLA) specification as an example aspect language for service contracts and demonstrate the facility with which AOP techniques allow web service management code can be implemented within an existing distributed information flow programming framework.

The Infosphere project has already created a toolkit to support the specification and generation code for information flow applications [3]. The basic Infopipes toolkit consists of a high-level specification language, Spi; an XML-format of this language, XIP; and the Infopipe Stub Generator, or ISG, which generates the communication code. When using our framework and AOP techniques, we find it *is* significantly easier to augment our existing specification language with AOP, although it is still non-trivial. Doing so, allows us to add support for WSLAs and to generate code supporting web service agreements. Three primary factors contribute to this: First, DSLs such as XIP typically capture recurring designs, so a weaver capable of using the underlying design pattern can reduce the number of extraneous joinpoints and thereby reduce complexity in writing new aspects. Second, XIP is defined over XML and it benefits from XML's inherent extensibility and maturing software tools such as parsers and XSLT pattern processors, facilities which have proved to be very useful for incorporating aspects. Third, the design of our generator as a processing pipeline aided us by easing the insertion of code weaving logic as an additional processing component.

The first contribution of this paper is a concrete implementation of an experiment, using our AXpect AOP weaver to implement WSLA support for an application created using the ISL/ISG framework. The weaver processing

Manuscript received February 9, 2004. This work was done as part of the Infosphere and Embedded Infopipes projects, funded by DARPA through the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs. The research was also partially funded by NSF's CISE directorate, through the ANIR and CCR divisions as well as the ITR program. In addition, the research was partially funded by Intel.

G. S. Swint is a Ph.D. graduate student in the College of Computing at the Georgia Institute of Technology, Atlanta, GA 30332-0280 USA (phone: 404-385-2585, e-mail: swintgs@acm.org)

C. Pu is a Professor, holds the John P. Imlay Chair in Software, and is the Co-Director for the Center for Experimental Research in Computer Systems in the College of Computing at the Georgia Institute of Technology, Atlanta, GA 30332-0280 USA (e-mail: calton@cc.gatech.edu).

component accepts AOP-augmented XIP from our code generation component and uses XSLT to retrieve data from the application's XIP specification and from its contract WSLA. It then inserts WSLA support at the source level.

The second contribution of this paper is an experimental evaluation of the AXpect weaver through a non-trivial information flow application. In our experiment, we used WSLA to apply resource constraints to the application and created a cooperative image sender and receiver services that respect CPU usage requirements of the receiver end through sender adaptation. Our framework allowed us to implement the constraints and monitoring in an incremental fashion. So far, AXpect has generated and woven about 30% of the final code.

The organization of this paper is as follows: Section II provides a motivating example and background material relevant to Infopipes. Section III describes the implementation of the AXpect weaver for AOP support in Infopipes. Section IV then illustrates the use and results of AOP and the AXpect weaver in the context of our motivating example. Finally, Sections V and VI, respectively, provide related work and the conclusion.

II. Motivation and Background

A. Motivating Example

We will use a distributed image streaming application to illustrate the functions performed by the Spi/ISG toolkit and then demonstrate the use of a WSLA document plus the AXpect weaver to impose resource constraints and add adaptation to the image source service. For this application, we have a sender, the source of our images, and a receiver, which contracts to accept and process the image data stream. In the base implementation, the sender transmits images to the receiver at an unconstrained rate. We then use a WSLA document to add a CPU usage constraint so that the receiver measures the resource usage of the sender and returns the data to the sender so the sender can adjust its sending rate.

Operational requirements of this simple sender/receiver application are common to many distributed information flow applications. For its part, the receiver must create a socket, publish connection information, and wait for an incoming connection. The sender follows a complementary series of steps. It creates a socket, looks up the receiver's connection information, and then establishes the socket connection. In the steady state, the sender transmits data to the receiver through the connection. In the base implementation, however, it is easy for an overeager sender to swamp the receiver with too many images and thereby use up a disproportionate amount of receiver resources which may be needed for other tasks. We discuss how a WSLA allows the sender and receiver to manage this situation in the experimental section.

B. ISL, ISG and Infopipes

A great deal of code in information flow applications pertains solely to creating, using, and maintaining the communication connection. This code implements functionality such as connection establishment and code to marshall and transmit data and following that to receive and unmarshall data. We exploited these commonalities and developed the Spi language and ISG code generator for specifying and generating communication frameworks to support these applications.

The Spi/ISG toolkit has two parts: the domain specific language, known as Spi, and a code generator, the ISG. Spi captures application design by describing each transformational step as a "pipe" which has inputs, outputs, and some function that maps between them. Spi also allows a developer to specify datatypes used for communicating between processes.

Once a Spi description is created, it is compiled into an intermediate representation – the *XIP* (XML for Infopipes) description. Our compiler for Spi is based on the Ply lexer/parser package. The compilation of Spi is a straightforward transformation of the datatype, pipe, and connection information to XML structures which is then augmented with some configuration information.

XIP is itself another domain specific language though not designed for human readability. Rather, XML syntax is most suited as input to programs. XIP serves as input to the Infopipe Stub Generator (ISG), our software that generates the communication stubs for handle connection establishment, data marshalling, etc. Figure 1 provides a visual overview of the ISG architecture including the AXpect weaver (which we will discuss in more detail in Section III). Overall, it is similar to the application code generator architecture described in [4]. The XIP specification is the input, and C source code accompanied by makefiles are the output of the code generation process.

Code generation proceeds through two phases. First, the XIP specification is passed to the actual code generation

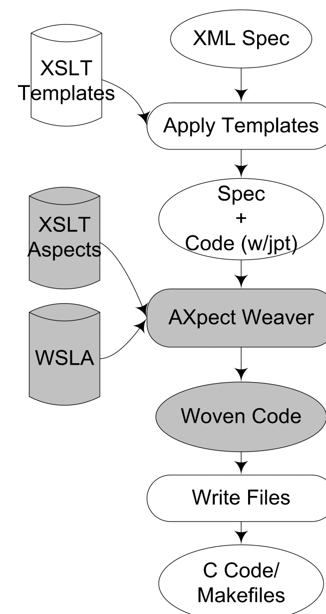

Figure 1. ISG with support for AXpect weaving.

component, in our case a collection of XSLT-based templates. These templates are XSLT statements embedded within a C code skeleton which can retrieve relevant data from the XIP specification, as the `<xsl:value-of>` element in Figure 2 does. The result of applying the XSLT to the XIP document is a new XIP document containing both the original specification and also the generated C code. In the second phase, this new XIP document is passed to the final stage of the stub generator. Here, the XIP is cracked into individual files, stripped of XML, and written into directories.

In our example application, the developer first creates the Spi description which defines two pipes – a sending pipe and a receiving pipe. These pipes are connected, and the sender transmits image data as type `ppm`, which is a small header followed by a byte array, to the receiver. This Spi specification is compiled into an equivalent XIP, which is then passed to the ISG. The ISG selects the proper templates and then writes the files into the proper directories. The developer can then add the C source code implementing the core functionality of the sender and receiver.

III. ASPECT SUPPORT IN THE ISG

To the toolkit described in the previous section, we have added support for AOP. This topic we divide into three areas and discuss them in this section. We address, first, aspect specification, in which we designate the aspects to be woven with our base code; second, weaver support in the templates; and third, a method to implement each aspect and specify code, advice, and pointcuts.

On top of the basic streaming application described in the previous section, we added support for resource management using AOP to evaluate the AXpect weaver. Normally, each resource constraint (e.g., CPU or network usage guarantees) needs specific code that touches several parts of the system, since resource management requires end-to-end cooperation. For example, performance is limited by the bottleneck component in the entire system and security is limited by the weakest component. In this case, we use CPU usage monitors and an additional feedback channel to allow the sender to respect resource constraints of the receiver by adapting its sending rate.

```
int <xsl:value-of select="$thisPipeName"/>( ) {
  <jpt:pipe point="user-declare">
  ; // USER DECLARES VARS HERE
  </jpt:pipe>
  <jpt:pipe point="user-function">
  ; // USER CODE GOES HERE
  </jpt:pipe>
  return 0;
}
// startup all our connections
int infopipe_<xsl:value-of
               select="$thisPipeName"/>_startup()
{
  <jpt:pipe point="startup">
  // start up outgoing ports
  // <xsl:for-each select="./ports/outport">
  infopipe_<xsl:value-of
               select="@name"/>_startup();
      </xsl:for-each>

  // start up incoming ports
  // <xsl:for-each select="./ports/inport">
  infopipe_<xsl:value-of
      select="@name"/>_startup(); </xsl:for-each>
  </jpt:pipe>

  return 0;
}
```

Figure 2. Excerpt from a template that generates connection startup calls and skeleton for the pipe's function. Line breaks inside XSLT tags do not get copied to the output.

A. Aspect Specification

The developer of an application must indicate to the weaver what aspects are to be applied and on which components to apply them. We chose to implement aspect support at the XIP level and postponed the research question of aspect specification in Spi. The decision stems from two main reasons. First, XIP-level aspect specification is required in any case, since Spi is translated into XIP. Second, there is no standard WSLA specification language – competing standards include CDL from the QuO project [5], and proposals by HP [2] and IBM [1].

Adding aspect statements to each pipe specification that generates code is done by adding an XML element which carries the name of the aspect and any additional information the aspect requires. For instance, if we wish to apply an aspect to the receiver that generates rate controller functionality and it references a WSLA, then we write this:

```
<apply-aspect name="rateController.xsl"
doc="uav.xml"/>.
```

Aspects may specifically rely on functionality located in other aspects implying at least a partial ordering. Developers can denote this in the specification by nesting aspect application elements within one another, and the weaver will apply the most deeply nested aspects first. In our sender-receiver example we want to apply a CPU usage monitor which relies on the timing information generated in the timing aspect. In XIP the requirement is expressed like this:

```
<apply-aspect name="cpumon.xsl">
  <apply-aspect name="timing.xsl"/>
</apply-aspect>.
```

The AXpect weaver does not require a developer to specify dependencies between all the aspects in use. Alternatively, the developer can simply list the statements as children of the `<pipe>` element or of an `<apply-aspect>`. In these cases, the aspects are applied in the order listed.

B. Aspect Support in the Templates

C, our target implementation language, does not have a complete and robust aspect weaver for it, yet, as AspectC is still in development. On the other hand, we also do not need the full power of a general aspect weaver, either, because we limit ourselves to a specific domain – information flow applications. Instead, we adopted an approach analogous to the explicit programming model of ELIDE [6]. In AXpect, joinpoints are XML elements explicitly written into the templates and they are selected via XPath pointcuts. Unlike ELIDE, however, we can limit the number of joinpoints explicitly expressed since we are in a specific domain and

have already identified some repeated patterns to express through them. For instance, we know that each pipe at run time proceeds through three main phases: startup, running, shutdown. Furthermore, we even know what specific substeps are required for each of those phases, e.g. initialization and location advertising by a socket inport during startup.

Knowing these steps, we can insert XML elements that explicit denote the boundaries of execution for each step and sub-step. Referring back to Figure 2, we see XML elements denoting the startup of a pipe (`<jpt:pipe point="startup">`) and delineate the group of calls to start up each inport or outport, and another element (`<jpt:pipe point="user-function">`) denoting the position where the user will insert the code to do the work of the pipe. Note that we place the tags inside the functions so that code can be added inside the function. We have additional tags outside the functions that denote the contents of a source file or header file so that an aspect developer can affect application structures larger than a single function. In fact, an aspect can also add entirely new files to the suite of generated files and generate calls to functions in the new files.

XSLT was designed for manipulating and creating XML documents. Therefore, the presence of XML tags in the generation templates is not unusual and requires no extra "workarounds" when coding templates nor does the supplemental XML reduce or break the functionality in the base templates. For the most part, code included in a template or aspect needs very little modification. Typically, C code only requires ampersands and less-than symbol conversion.

We placed the joinpoint tags in a separate namespace. This segregates the joinpoint designators from tags that may have other purposes, such as denoting file information for the code generator or XSLT elements.

C. Implementing an Aspect

An aspect for the AXpect weaver is an XSLT document. The collection of pointcuts, advice, and code that constitutes an aspect are expressed using a collection of `<xsl:template>` statements in a stylesheet and use the template's `match` attribute to execute the pointcuts through XPath statements. For instance, the XPath query to select all instances of the joinpoint "middle-module" would look like this: "`//jpt:pipe[@point='module']`".

Most of the time, we apply an aspect to only one pipe in a collection of all the pipes generated to create a system. To select the code for only that pipe from the entire collection in the XML document we can condition our XPath statement to narrow the selection of joinpoints in the desired fashion. Since this is such a common requirement, we extract the identify the pipe from the `<apply-aspect>` statement then hand pass it to the template through variables. Then, if you wanted only the `middle-module` template for the `receiver` pipe, you qualify your selection statement with XPath attribute selectors:

```
"//filledTemplate[@name=$pipename]
  [@inside=$inside]//
      jpt:pipe[@point='module']"
```

where `$pipename` and `$inside` are variables. We can restrict join point selection by adding XPath predicates, or select multiple joinpoints with the '|' (OR) operator as well.

AspectJ has three advice keywords, `before`, `after`, and `around`, direct the weaver to run the aspect code before, after, or in the `around` case, either both before and after or instead-of the joinpoint. When around is used for before/after semantics, the developer controls the execution of the code in the joinpoint with the `proceed` keyword. Rather than designating advice with a keyword, advice with AXpect is expressed through the structure of the XSLT template. Once the XPath statement selects the joinpoint, then the aspect writer must explicitly copy the joinpoint and its code to the output using the `<xsl:copy>` element. At this point, the aspect writer can choose to add code before, after, around, and instead-of (by omitting the copy directive) the joinpoint. In Figure 3, we can see that the timing aspect inserts code around the middle function of a pipe. With this approach, each piece of inserted code has access to the same variables and stack context as the original code.

An aspect may also need to refer to data in the XIP specification. Since the specification is presented along with the code for weaving, the aspect code can refer to the specification in a same manner that templates retrieve the data by using `<xsl:value-of>` element.

When developing an aspect, a writer may wish to include joinpoints in the aspect code to augment the joinpoints available in the base templates and thereby expand the joinpoint space. Doing so provides hooks for aspects which may be applied later, and allows the later aspect code to take advantage of the earlier aspects' functionality. This denotation is accomplished in the same fashion as adding joinpoints to the original templates -- by adding XML elements to the aspect template in the same fashion as for the code generation templates. In Figure 3, it is easy to see the added joinpoint `<jpt:time-process>` which denotes the block of code in the aspect related to timing the execution of the function of a pipe. Note that we kept the variable declarations outside the

```
<xsl:template
  match="//filledTemplate[@name=$pipename]
              [@inside=$inside]//jpt:pipe-middle">

  struct timeval base;
  struct timeval end;

  <jpt:time-process>
  // take timing here
  gettimeofday(&amp;base,NULL);

  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>

  gettimeofday(&amp;end,NULL);
  usec_to_process = (end.tv_sec - base.tv_sec  ) *
               le6 + (end.tv_usec - base.tv_usec);
  fprintf(stdout,"Time to process: %ld\n",
                              usec_to_process);
  </jpt:time-process>
</xsl:template>
```

Figure 3. An excerpt from `timing.xsl`, the timing aspect implementation.

joinpoint element. This is to comply with the C requirement that variables be declared at the top of a code block. Following this convention helps ensure that the aspect code does not introduce syntax errors.

Finally, for WSLAs, we must retrieve the constraints encoded in the WSLA and incorporate them into the generated code. We do this by utilizing XSLT's "`document`" function which allows for a developer to operate from any XML data source. In our test we have found it simplest to simply define a variable that represents the root of the XML document, and then this can be re-used throughout the aspect.

D. The AXpect Weaver

The AXpect weaver is the component of the ISG that brings together the preceding three topics by interpreting aspect specification statements in the XIP, loading the implemented aspects from disk, and applying them to the template-generated code. The modular structure of the ISG allowed us to insert the AXpect engine as a processing stage executed after applying the XSLT code generation templates, as shown in Figure 1. After generation, the produced pre-code is re-bundled with the description as an XML document and passed along to the AXpect weaver. The weaver itself is a C++ component that uses the Xerces-C XML parser and Xalan-C XSLT processor to resolve, load, and weave the aspects.

Generally, the process for weaving aspects is straightforward, as the complications of finding joinpoints, executing pointcuts, and weaving are in the XSLT engine.

Weaving proceeds recursively through the following steps on each pipe:

1. Retrieves the first `<apply-aspect>` element from the specification.
2. If the aspect contains more `<apply-aspect>` statements, then the AXpect applies those aspects first, and re-enters the process of weaving at this step.
3. The weaver retrieves the aspect code from disk (aspects are kept in a well-known directory).
4. Apply the aspect to the code by passing the aspect XSLT stylesheet, the generated code with joinpoints, and system XML specification to the Xalan-C XSLT processor. The result is a new XIP document that again contains the specification, woven code, and joinpoints that were retained or even added by the aspect.
5. The weaving result serves as input for any aspects that follow the current aspect. This includes aspects which depend on the current aspect's functionality, or functionally independent aspects that are simply applied later.
6. Once all aspects have been applied, then the entire XML result document is passed to the last stage of the generator to be written to disk. Any residual XML joinpoints in the woven code remain until the last stage removes them as the code the generator writes the source files to disk.

IV. USING AXPECT

A. Scenario

In the base implementation of the streaming image application, a sender transmits images to the receiver as quickly as possible given network conditions and the senders own computational load. However, in some environments it is desirable for the receiver to dictate a limit on the rate at which the sender transmits data. For instance, the receiver may wish to perform compute-intensive transformations on the data, or the receiver may be collecting images from multiple sources (possible even from multiple network segments) at the same time. In such cases, it is useful for a rate limiter to be programmed into the sender which responds to receiver issued WSLA information messages about CPU use.

For the base scenario, there are a total of fourteen files generated each for the sender and receiver:

- `sender.{c,h}` or `receiver.{c,h}`: the datatype declarations, the middle function of the pipe, its startup, and its shutdown code
- `ppmOut.{c,h}` or `ppmIn.{c,h}`: header files for the communication functions and source files implementing marshalling, communication, and connection establishment
- `runtime.{c,h}`: header and library functions for support of connection establishment. There is one of each of these files for the sender and receiver.
- a `Makefile` for each sender and receiver.

When the application runs, it first calls the startup code for the pipe. This in turn calls the startup code of each connection for opening and connections. Once startup is complete, the pipe enters its main running phase, which consists acquiring data and submitting it to the communication layer in a continuous loop. The communication layer then manages the network transmission. Communication is asynchronous between the sender and receiver.

B. Implementation

Our base scenario simply allows the sender to transmit data unchecked to the receiver using the base code generated by the templates. To add rate-limiting functionality, then, to the base implementation requires the following changes to a base sender-receiver implementation:

- Add support for resource metrics to the receiver reevaluated each time the receiver processes an application packet. Requires code added to the receiver when it initializes and when it processes each packet.
- Add a reverse channel for WSLA information messages from the receiver to the sender. This requires discovery and connection code on the client and receiver plus a mechanism to multiplex and demultiplex control messages.
- Add rate CPU metric code to the receiver which marshals and sends informational messages to the sender about the observed metric under a chosen reporting policy (e.g. windowed vs. un-windowed). It builds on the functionality of the cpu monitor aspect and the control channel. So, those aspects must be present first.
- Add rate control code to the sender. This code must retrieve messages from the control channel, demultiplex them, and

behave appropriately. It again depends on the control code being applied first. The sender "throttles back" by sleeping after image transmission if the receiver reports greater than 20% CPU usage.

Figure 4 illustrates the application and aspects. Note that in addition to crosscutting the base design of the application, several aspects crosscut other aspects.

Implementing the aspects proceeded in several steps. First, since we had not previously used aspects with our template code, we added a total of 18 joinpoints to the base template code. For the most part, these additions corresponded to each major syntactic or logical unit of code. For instance, we mark the header and implementation sections for the communication code and function of the pipe, the body of the pipe startup code, the body of the pipes middle function, the code that reads the socket, and finally, we also add a joinpoint (`<jpt:make-rule>`) to the `Makefile` for the sender and receiver.

For adapation in the application, we created six aspect files. On the sender side we used two aspects: `control_sender`, which implemented the sender-side control channel, and `sla_sender`, the implementation of the sender's response to receiver rate requests. On the receiver side we used five aspects: `timing`, which provided base timing measurements for CPU usage computation, `control_receiver`, an implementation of the receiver-side control channel, `cpumon`, which monitored CPU usage, and `sla_receiver`, which sent rate messages to the sender. Each aspect corresponded to one XSLT file.

When creating the control channel, we placed the bulk of the functionality in files separate from code generated for the base implementation. It added startup code and `make` rules to the sender and receiver output files. Altering the `Makefile` allowed automatic compilation of the extra files for the pipes, and adjusted the compile and link flags by adding required libraries like `-lpflags` for supporting the separate thread of the control channel.

For an illustration of how disruptive even relatively simple additions can be to the application, we will explore the modifications on the sender side more closely. First of all, the sender must establish a control. Since we do not want our control channel to interfere with the main communication of the application, we place the service for the control channel in a separate thread. This means that at pipe startup, we must create a separate thread, create a socket within that thread, and connect to the receiver's control socket. Next, we add support for the rate control; therefore, the startup code must also initialize rate information variables. In this case, this entails setting the sender's sleep flag and guard variable to 0 (the guard variable allows us to turn off the throttle control if the sender is allowed to send at its maximum rate). In addition to this startup complexity, the rate control aspect inserts into the control channel's demultiplexing function code that routes incoming control information to the `"rateCmdReceived()"` function, which takes proper action. Furthermore, to actually implement the rate throttling, the rate control aspect inserts a guard statement and `usleep()` call after the Infopipe completes its data transmission. Each of these changes involves installing variables at various scopes (global, module, and local) and in multiple header files. Finally, since we add new files to the application, we also insert the aforementioned `Makefile` rule and add the corresponding object files and flags to the link list.

C. *Results*

After applying each aspect, we saved the produced XIP document and then stripped the XML, comment lines, and whitespace-only lines. This yielded a monolithic document that contained the source for the entire distributed system equivalent to a concatenation of the generated files minus whitespace and comments. We then measured the number of non-comment lines of code (NCLOC) added by each aspect. For a sample of woven code, see the Appendix which contains an excerpt from the receiver’s pipe middle.

Table 1 presents the lines of code added by each aspect. The column "Where" denotes whether the aspect applied to code generated for the sender or the receiver, and the "Lines Added" metric is the number of non-comment lines added. Generally, the aspects add C code, but in the case of the aspects pertaining to the control channel, `control_receiver` and `control_sender` new rules for `make` were woven in as well.

Table 2 and Table 3 detail exactly which files were affected by the aspect being woven. For the files `control` and `sla` files, we place the X in parentheses since these files were *created* by the respective aspect. We can see that the number of files altered by each aspect varied greatly. In a complementary fashion, we also note that each file may be affected by a variable number of aspects, too. Another observation we may make adding a given functionality may not necessarily affect each side of the application symmetrically.

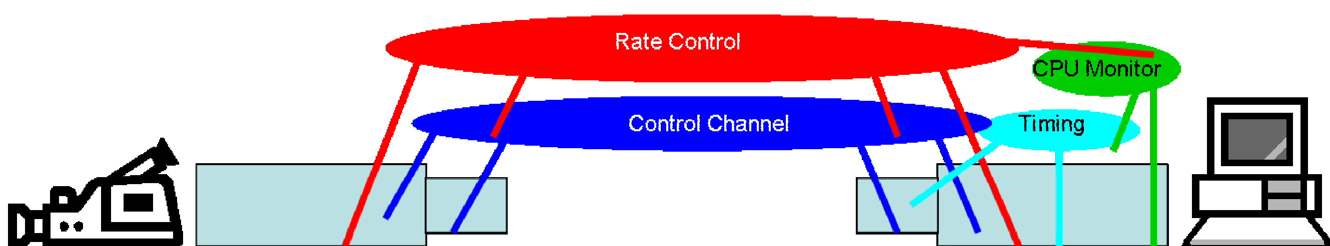

Figure 4. The image stream application with rate controlling aspect.

TABLE 1
NCLOC ADDED

Aspect	Where	Lines Added
control sender	sender	117
sla_sender	sender	73
timing	receiver	50
control_receiver	receiver	125
cpumon	receiver	14
sla_receiver	receiver	55
Total from aspects		**434**
Base Implementation		**976**
Base + Aspects		**1410**

TABLE 2
SENDER-SIDE FILES AFFECTED.

Aspect / Affected File	Makefile	sender.h	sender.c	PpmOut.h	PpmOut.c	control.h	control.c	sla.c	sla.h
control_sender	X		X		X	(X)	(X)		
sla_sender	X		X			X	X	(X)	(X)

TABLE 3
RECEIVER-SIDE FILES AFFECTED.

Aspect / Affected File	Makefile	receiver.h	receiver.c	ppmIn.h	ppmIn.c	control.h	control.c	sla.c	sla.h
timing			X		X				
control_receiver	X		X		X	(X)	(X)		
cpumon			X						
sla_receiver	X		X			X	X	(X)	(X)

V. RELATED WORK

From a DSL standpoint, Spi/XIP may be compared to Spidle and Streamit. However, Spidle is oriented towards synchronous, single-process applications [9], and StreamIt aims for streaming DSP applications and processors with grid based architectures [10], [11].

Other AOP projects have taken advantage of DSL patterns various ways. For instance, Bossa applies AOP concepts to scheduling in the kernel [12]. It defines a limited number of pointcuts in the kernel code and then uses an event based AOP model to implement the aspects. However, Bossa does not actually add AOP to a DSL. Instead, Bossa takes the view that the DSL actually implements an aspect describing a single aspect, scheduling, of the Linux kernel. Also, the developers of the ACE+TAO orb used aspect oriented and DSL techniques to expose the real-time functionalities of their orb with contract objects and associated Contract Description Language (CDL) [5]. CDL is limited to monitor and control functions only. However, they then used a second DSL, the Aspect Structure Language (ASL), for applying aspects that mediate interactions between distributed objects [13]. ASL, however, recognizes only a few types of pointcuts that are specific to CORBA development, and application developers can not extend the joinpoint space.

In AOP, XML has been used to capture and manage aspects in the requirements phase [7]. Schonger *et al* in [8] proposed XML as a generic markup for describing the abstract syntax trees of general purpose languages and the concept of creating AOP *operators* for weaving. We agree with their observation that XML aids AOP experimentation; however, they did not explore the use of XML and XSLT use with domain specific languages and code generation.

ELIDE is an AOP extension of Java that allows developers to add explicit, named pointcuts at any point in their programs [6]. It differs from our approach in two respects: first, ELIDE is a Java-specific language extension whereas AXpect works on C and uses generic, pre-existing XML and XSLT tools. Second, ELIDE is general purpose mechanism whereas AXpect is targeted to our domain-specific stub generator and distributed, streaming applications.

Of course, AspectJ and AspectC also implement aspect weavers [4] [14], but their weavers and pointcut declarations are closely tied to the implementation structure of the application; therefore, changes to the original source code may break the aspects. AXpect relies instead on explicitly denoted functionality and should be somewhat more robust in that regard.

Finally, Gray *et al* in [16] propose that AOP techniques be used in specific domains at the level of the domain abstraction as well as at the implementation level. They propose the Embedded Constraint Language (ECL) for creating new domain-specific weavers that process domain models and not implementation source code. This is in contrast to AXpect, which is an implementation level weaver.

VI. CONCLUSION

We have demonstrated the ease with which WSLA support can be added to an existing domain specific framework using AOP and explicit programming techniques to an existing. Using the AXpect weaver, we build an example application that uses resource constraints from a WSLA document to cooperatively manage resource usage through adaptation.

Our AOP framework entails aspect specification using WSLA to parameterize an aspect, and then use XSLT and XPath to write pointcuts and advice that operate on joinpoints explicitly denoted in our code generation templates. Furthermore, our code weaving occurs in C source code, a language which does not yet enjoy robust AOP support for the general application space.

Explicit joinpoints are feasible because our templates are in XSLT and because we operate in the specific domain of information flow applications. Since our aspects are also coded using XSLT, we can efficiently layer aspects to build on functionality and further modularize aspect development. Finally, XSLT allows us to retrieve data from the XML-based WSLA specification document at "no extra cost" through its support for multiple XML source documents.

APPENDIX

Below is sample code from the receiver side pipe function. To highlight the aspect code, we have replaced the application

code with ellipses. It shows the additional include statements, timing code, and call to evaluate the SLA metrics:

control aspect (control_receiver.xsl)
timing aspect (timing.xsl)
cpu usage metric aspect (cpumon.xsl)
SLA evaluation aspect (sla_receiver.xsl)

```
#include "receiver.h"
#include "ppmIn.h"
#include "control.h"
#include <sys/time.h>
#include <stdio.h>
extern long usec_to_port_startup;
extern long usec_to_port_shutdown;
extern long usec_to_recv;
long usec_to_pipe_startup;
long usec_to_pipe_shutdown;
long usec_to_process;
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
float CPUUsage;
static long lastUTimeUse = 0;
static long lastSTimeUse = 0;
static struct rusage usingNow;
#include "sla.h"
int receiver( ) {
  ; // USER DECLARES VARS HERE
.
.
  struct timeval base;
  struct timeval end;
  gettimeofday(&base,NULL);
  ; // USER CODE GOES HERE
.
.
.
  gettimeofday(&end,NULL);
  usec_to_process = (end.tv_sec -
          base.tv_sec  ) * 1e6 + (end.tv_usec -
          base.tv_usec);
  fprintf(stdout,"Time to process: %ld\n",
          usec_to_process);
  getrusage( RUSAGE_SELF, &usingNow );
  CPUUsage = ((float) usingNow.ru_utime.tv_usec +
        usingNow.ru_stime.tv_usec - lastUTimeUse +
          ((float)  usingNow.ru_utime.tv_sec +
                    usingNow.ru_stime.tv_sec -
                             lastSTimeUse) * 1.0e6)
            / (usec_to_recv + usec_to_process);
  lastUTimeUse = usingNow.ru_utime.tv_usec +
                        usingNow.ru_stime.tv_usec;
  lastSTimeUse = usingNow.ru_utime.tv_sec +
                        usingNow.ru_stime.tv_sec;
  fprintf(stdout, "Use pct %0.2f.\n",
                                  CPUUsage * 100);
  processSLA();
    return 0;
}
```

ACKNOWLEDGMENTS

This work is partially funded by DARPA/IXO as a project in the PCES program, by DoE as a project in the SciDAC's Scientific Data Management Center, by NSF/CISE as a project in the CCR division's Distributed Systems program, IIS division's Data and Application Security program, and the ITR program. Also, we would like to thank Charles Consel of INRIA/ENSEIRB (Bordeaux, France),

REFERENCES

[1] M. Debusmann, and A. Keller, “SLA-driven Management of Distributed Systems using the Common Information Model,” *IFIP/IEEE International Symposium on Integrated Management*. 2003.

[2] A. Sahai, S. Graupner, V. Machiraju, and A. van Moorsel, “Specifying and Monitoring Guarantees in Commercial Grids through SLA,” *Third International Symposium on Cluster Computing and the Grid*. 2003.

[3] G. Swint, C. Pu, and K. Moriyama, “Infopipes: Concepts and ISG Implementation,” *The 2nd IEEE Workshop on Software Technologies for Embedded and Ubiquitous Computing Systems*, Vienna, Austria, 2004.

[4] S. Sarkar, “Model Driven Programming Using XSLT: An Approach to Rapid Development of Domain-Specific Program Generators,” www.XML-JOURNAL.com. August 2002.

[5] J. P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, and K.R. Anderson, “QoS Aspect Languages and Their Runtime Integration,” *Proceedings of the 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (LCR98). Pittsburgh, Pennsylvania, May 28-30, 1998.

[6] A. Bryant, A. Catton, K. de Volder, G. C. Murphy, “Explicit programming,” *1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, 2002.

[7] A. Rashid, A. Moreira, and J. Araújo. “Modularisation and Composition of Aspectual Requirements,” *1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, 2002.

[8] S. Schonger, E. Pulermüller, and S. Sarstedt, “Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees,” *Proceedings of the 2nd German GI Workshop on Aspect-Oriented Software Development* (In: Technical Report No. IAI-TR-2002-1), University of Bonn, February 2002, pp. 59 – 64.

[9] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. “Spidle: A DSL Approach to Specifying Streaming Applications,” in *Proceedings of the Second International Conference on Generative Programming and Component Engineering*. LNCS 2830, September 22-25, 2003, pp. 1-17.

[10] W. Thies, M. Karczmarek, M. Gordon, D.Z. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe. “A Common Machine Language for Grid-Based Architectures,” *ACM SIGARCH Computer Architecture News*. New York, June, 2002, pp. 13-14.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe. “StreamIt: A Language for Streaming Applications,” in *Proceedings of the 2002 International Conference on Compiler Construction*, LNCS, Grenoble, France, April, 2002.

[12] L.P. Barreto, R. Douence, G. Muller, and M. Südholt, “Programming OS Schedulers with Domain-Specific Languages and Aspects: New Approaches for OS Kernel Engineering,” *International Workshop on Aspects, Components, and Patterns for Infrastructure Software* at AOSD, April 2002.

[13] BBN Technologies. QuO Toolkit Reference Guide. 2001.

[14] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. “Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code,” in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*, Vienna, Austria, 2001, pp. 88-98.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. "An Overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Budapest, Hungary, June 18-22, 2001, pp. 327-353.

[16] J. Gray, T. Bapty, S. Neema, D.C. Schmidt, A. Gokhale, and B. Natarajan, “An Approach for Supporting Aspect-Oriented Domain Modeling,” in *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, LNCS 2830, September 22-25, 2003, pp. 151-168.