DSL Weaving for Distributed Information Flow Systems

Calton Pu, Galen Swint

CERCS, College of Computing, Georgia Institute of Technology, 801 Atlantic Drive,
Atlanta, Georgia, 30332-0280 USA
calton@cc.gatech.edu, swintgs@acm.org
http://www.cc.gatech.edu/projects/infosphere/

Abstract. Aspect-oriented programming (AOP) is a promising field for reducing application complexity. However, it has proven difficult to implement weavers for general purpose languages. Nevertheless, we felt some functionality for our information flow abstraction, Infopipes, might be best captured in aspects. In this paper, we describe a weaver built for domain specific languages (DSLs) related to Infopipes around an off-the-shelf XSLT processor. Aspects are written in XSLT, XML annotations are added to existing DSL generation templates, and XML directives are added to our Infopipes specification. Finally, we successfully demonstrate a generated+woven application that adds the quality of service (QoS) dimension of CPU usage awareness to an image streaming application.

1 Introduction

Web services are gaining momentum in industry as a paradigm for building and deploying applications with a strong emphasis on interoperability between service providers. Inherent in this movement is the need to codify and monitor performance of applications or application components which are administered or purchased from another party. This has led to the recognition and proposal of service level agreements (SLAs), which can capture expectations and roles in a declarative fashion [1,2]. One view of such agreements is that they constitute a domain specific language. As with any language, then, the question becomes how to map the "high" abstraction of the SLA language into a lower-level implementation. This amounts to run-time measurement, feedback, and adaptation interspersed into a web service-enabled application.

In addition to viewing SLAs as a domain specific language, it is helpful to consider them as an aspect of a web-based application in the sense of Aspect Oriented Programming (AOP)[3]. This follows from noting that SLAs typically describe some application functionality that *crosscuts* application implementation which means that given a complete implementation of the application including service monitoring, then the SLA implementation code will be found in multiple components of the main application, and furthermore, the crosscutting code is heavily mixed, or *tangled*, in components where this crosscutting occurs.

AOP centers on the use of source code weavers to attack this problem crosscutting an tangling in an organized fashion. Currently, the most significant AOP tool has been the AspectJ weaver [4], developed after several years of effort, which supports the addition of

aspect code to general Java applications. Applying the same techniques to C and C++ code, however, has been harder. The question arises, then, as to whether it is difficult to implement weavers for any language.

We built the AXpect weaver into the existing code generation framework, the Infopipe Stub Generator [5,6]. The ISG has three important parts: the intermediate representation, XIP; a repository of types and Infopipe descriptions; and a collection of templates written in XSLT.

This paper describes the architecture and implementation of the AXpect weaver in detail, as well as discusses a prototypical example application whereby a WSLA document is used to specify CPU usage policy between a server and client of a media stream. In section 2, we introduce the Infopipes abstraction for distributed applications. In section 3, we discuss the pre-existing code generator for our project, the ISG. In section 4, we present a close look at how we implement AOP in our system, and in section 5, we evaluate the weaver in the context of an example media application.

2 Infopipes

It has been long-recognized that RPC, while promising, has problems as a distributed programming paradigm. This mainly stems from the fact that a distributed application may have to deal with comparatively vast time scales, less security, and much greater divergence in resource availability than when operating on a single machine, even if it is a parallel machine. Consider that memory access and procedure call times may be measured in nano- or micro-seconds, but that web applications must address millisecond latencies – three to six orders of magnitude longer.

Infopipes are designed to take these differences into account, particularly for information flow applications. One reason for targeting information flow applications is that they are difficult to capture abstractly using RPC because their normal operation, sending a continuous stream of data, is innately mismatched to RPC's request/response scheme. Second, such applications often involve multiple processing steps, a concept that is again not addressed by RPC's encouragement of the client-server style. Finally, RPC completely obscures the communication taking place in the application, so that if latency, bandwidth, security, or some other property is needed then a developer must "uncover" the communication and insert new code to recover lost data about connection performance, and add any new functionality by hand which may be particularly difficult if some sort of end-to-end property is desired to be enforced. As we stated before, Infopipes expose the communication step, and make it much easier for a developer to capture connection information and implement properties around needed communication boundaries.

The Infopipes architecture is service-oriented – it encapsulates granules of distributed computation which are intended to be composited together [7] – just like those proposed for web service applications. While the ISG does not currently explicitly support XML as

a wire format as is currently required to be Web Service compliant, it in no way excludes such a possibility, and even some previous unpublished Infopipe experiments have used XML as an *ad hoc* wire format. The ISG, in fact, already supports two completely different wire formats – PBIO, which is the wire format for ECho, and x86 byte-ordered data, as might come directly from a C program.

We have devised a prototype application to demonstrate Infopipes. The application is a video-streaming example in which the receiver of the video stream has Quality of Service requirements; it is constrained by its CPU resource and must provide feedback to the sender of the stream to control image arrival rate. Our code generator creates the communication setup, binding, and marshalling code and then automatically incorporates the QoS code which is parameterized in an external WSLA document. In the next section, we describe the implementation of our ISG to generate the base communication code. For this example, we will denote as our base application the sender and receiver's communication code with no QoS supporting code.

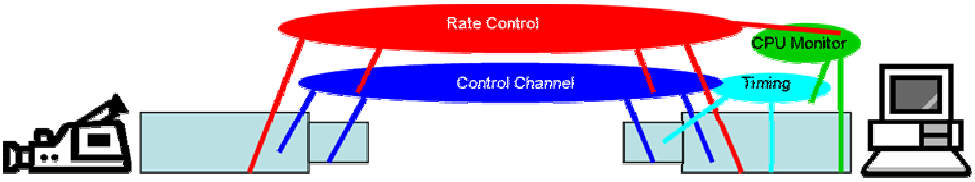

Fig. 1. The QoS-aware application.

We can see that there the application requires several functions to be implemented to support its QoS needs: a control channel, for feedback information; timing tags and code to monitor the CPU usage from times gleaned; and a rate control channel which implements the actions to take based on observations from the CPU monitor.

3 The ISG

The ISG toolkit has been developed for the Infosphere project to automate the programming of Infopipes code for developers. It consists of a human-friendly descriptive language Spi (Specifying Infopipes), an intermediate descriptive language XIP (XML for Infopipes), a repository for persistence of defined Infopipes structures, and a hybrid C++/XSLT code generation engine.

For a developer, converting a Spi specification to compilable source code is a three-step process:

1. Create a Spi document to describe the information flow system.
2. Compile the Spi into XIP.
3. The XIP is then processed with the ISG.

The ISG can read the XIP, and as shown in Fig. 2 below (which also includes the AXpect weaver), it proceeds through multiple stages to generate the communication code for the application:

1. The XIP is processed; new specifications go to the repository.
2. Previously defined specifications are retrieved and ISG constructs a full specification for a generated, termed a XIP+ document because it has a similar format, but is augmented with additional data.
3. Once the XIP+ document is completed, the XIP+ document is processed with our XSLT code generation templates. The result is another XIP+ document that also includes all the information from the source XIP+.
4. Code weaving can now be performed (see Section 4).
5. Code is written to directories and files ready for compilation.

The choice of XML for our intermediate format has proven to be beneficial even though XML's primary role is in the data connections between organizations. Instead of only inter-organizational data interchange, however, we use it for data interchange during the building of an application. This provides important advantages. First, it allows us to retain and add to semantic information that might otherwise be lost from stage-to-stage within the code generator. In particular, it allows us to capture domain information injected by virtue of operating in the Infopipes domain and with the Infopipes suite of domain languages. Such information is not readily preserved by general purpose programming languages. Second, it allows us to have one common wrapper format for multiple languages. Using XML, we can treat our code as data (which it is), and that fact allows us to modify the code after the generation phase. This technique is already widely

```
<pipe class="ImagePipelinePlain">
   <subpipes>
     <subpipe name="imagesSource"
              class="SendingPipe"/>
     <subpipe name="imagesReceive"
              class="ReceivingPipe"/>
   </subpipes>
   <connections>
     <connection comm="tcp">
       <from pipe="imagesSource"
             port="out"/>
       <to pipe="imagesReceive"
           port="in"/>
     </connection>
   </connections>
</pipe>
```

Fig. 3. Example XIP Infopipe specification.

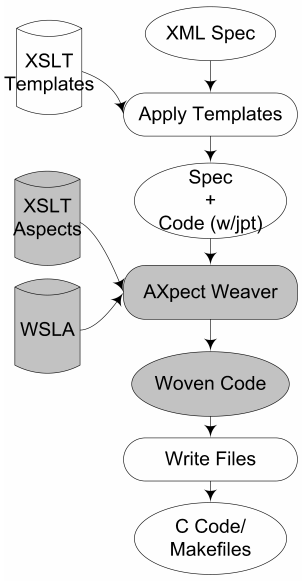

Fig. 2. ISG with support for AXpect

used in programming languages, but is only recently catching on in code transformation. Examples in general purpose languages include LISP macros, C++ templates, and Java generics. Each of those, in some fashion, allows the programmer to create a type of data-code hybrid. Later, as needed, certain parameters can be changed and custom code can be created for an application.

4 Weaving

Weaving is a concept from AOP in which new pieces of code are executed in a controlled and automatic fashion near, around, or instead-of code in a core application. We devised a source-level weaver to insert new functionality on top of video application. The weaving integrates code to measure CPU statistics, provide feedback, and adapt to changing transmission environment. It has a goal of maintaining CPU usage receiver-side below a given level.

There are three key concepts that enable the weaver. First, we attach semantic tags to the source code that we generate. Second, we use XSLT to execute the weaving of the aspect, and third, we insert weaving directives into the Infopipes description file.

Any weaver must have some points in the target code that it can identify. These are the "joinpoints." In our case, we benefit from our domain specific arena. Because of this, we know that specific activities occur within each Infopipe with known ordering. For example, we know that each pipe has a start-up phase that includes starting up each inport and each outport, resolving bindings and names, and actually making connections. During these initializations, our pipe may initialize data structures for each inport or outport. In the code generation templates, there is template code for each of these "common tasks."

AOP has three types of advice: before, after, and around. A developer chooses a joinpoint using a pointcut, and then designates by keyword whether aspect code should execute before, after, or around (which subsumes instead-of) the selected joinpoint. One interesting subtlety is that in AXpect the explicit XML tags denote a semantic block of code, and not just a single point. This most closely relates to AspectJ "around" semantics. Still, we retain the before and after capability of the weaving, without loss of "power." One could also view it another way, in which the XML opening tag marks "before," the closing tag marks "after," and taken together the tags make up "around" semantics.

For a concrete example, consider a fragment of template for generating C code Infopipes. This template excerpt generates a startup function for the Infopipe. The startup function name is based on the name of the Infopipe. The XSL commands are XML tags which have the `xsl` namespace prefix (like the element `xsl:value-of` which retrieves the string representation of some XML element, attribute, or XSLT variable). The added joinpoint XML tag is bolded, and it denotes the beginning and ending of the code block that implements the startup functionality. We have reverse-printed the joinpoint XML for clarity, and printed the C code in bold to distinguish it from the XSLT.

```
// startup all our connections
```

```
int infopipe_<xsl:value-of select="$thisPipeName"/>_startup()
{
  // insert signal handler startup here
  <jpt:pipe point="startup">
  // start up outgoing ports <xsl:for-each select="./ports/outport">
  infopipe_<xsl:value-of select="@name"/>_startup(); </xsl:for-each>
. . .
  </jpt:pipe>
  return 0;
}
```

Sometimes a joinpoint does not denote code but language artifacts that are needed for code to be written correctly. In following example, we see that we can denote the header file for an inport. This allows the C developer of new aspect code to insert new function definitions at the appropriate scope.

```
#ifndef INFOPIPE<xsl:value-of select="$thisPortName"/>INCLUDED
#define INFOPIPE<xsl:value-of select="$thisPortName"/>INCLUDED

<jpt:header point="inport"  pipename="{$thisPipeName}" portname="{$thisPortName}">
int drive();
// init function
int infopipe_<xsl:value-of select="$thisPortName"/>_startup();
int infopipe_<xsl:value-of select="$thisPortName"/>_shutdown();
void infopipe_<xsl:value-of select="$thisPortName"/>_receiveloop();
// data comes in to this struct
extern <xsl:value-of select="$thisPortType"/>Struct
          <xsl:value-of select="$thisPortName"/>;
. . .
</jpt:header>
#endif // Infopipe<xsl:value-of select="$thisPortName"/>INCLUDED
```

Joinpoints remain with the code until the files are written to disk. After the generation phase, they serve as signposts to the weaver and aspects. If we consider our first example, then after generation for the pipe called “imageReceiver” there is this startup code:

```
// startup all our connections
int infopipe_imageReceiver_startup()
{
  <jpt:pipe point="startup">
  infopipe_inp_startup();
  infopipe_inp_receiveloop();
  </jpt:pipe>
  return 0;
}
```

At this point, obviously, the code looks very much like pure C ready for compilation, and most importantly, we and the AXpect weaver know *what* the code does in the context of the Infopipes domain. Interestingly, we find that so far only about 26 joinpoints are necessary for quite a bit of flexibility with respect to actions we can perform on the generated code. These joinpoints have evolved into some broad categories as evidenced in **Table 1**, below.

“Language Artifacts” help a developer structure his code properly. “Data” joinpoints relate to the structures that hold incoming/outgoing data. “Pipe” joinpoints define actions that occur during the overall running of the pipe. Communication layer joinpoints are needed because it is common for these packages to need to perform initialization, set-up, or tear down functionality only once per-application start, and some applications may

need to build on this communication layer behavior or modify it. Last, we have joinpoints on the inports and outports themselves.

Table 1. Catalog of joinpoints in the C templates. These are expressed in a shorthand such that in the table below *type*:*point* equates to <jpt:*type* point="*point*"> in the XML/XSLT

Language Artifacts	Data	Pipe	Comm Layer	Inport	Outport
make:rule header:pipe source:pipe header:inport source:inport header:outport source:outport source:userdeclare	data:define data:initialize	pipe:userfunction pipe:startup pipe:shutdown	socket:socket socket:bind comm-startup comm-shutdown	inport:startup inport:read inport:unmarshall inport:callmiddle inport:shutdown	outport:marshall outport:push outport:startup outport:shutdown

The second ingredient of the AXpect weaver is an XSLT file that contains aspect code. Every AXpect file has two parts. First, the aspect has some pattern matching statement, written using XPath and utilizing the XSLT pattern matching engine, to find the proper joinpoint and the code to be inserted. This corresponds to the role of the pointcut in an AOP system like AspectJ. The pointcut in AXpect is an XPath predicate for an XSLT match statement in a fashion similar to this:

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

We can dissect the elements of the pointcut XPath statement:

`//filledTemplate[@name=$pipename][@inside=$inside]` – structure-shy specification to find a filledElement template, which is a block of generated code and predicates to narrow filled templates returned to one for a particular pipe.

`//jpt:pipe[@point='shutdown']` – a specific joinpoint

Instead of keywords like AspectJ, the AXpect developer uses placement. The actual joinpoint and its contents are copied over by XSLT's `xsl:copy` instruction. A simple aspect for AXpect looks like this (the C code is bolded for distinction from the XSLT):

```
<xsl:template match="//filledTemplate[@name=$pipename]
          [@inside=$inside]//jpt:pipe[@point='shutdown']">
  fclose(afile);
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

It is now easy to see how aspect code, pointcuts and joinpoints, and advice mesh. The pointcut, in reverse print, we have already discussed, and it is contained in the `match` attribute to the `xsl:template` element. We can see the C code to close a file (`fclose(afile)`) is located before the `xsl:copy` command, which means that it will be executed before the rest of the shutdown code. The `xsl:apply-templates` is boilerplate XSLT that ensures the processor continues to pattern match to all elements and attributes of the generated document that lie inside the joinpoint element. (It is our plan, in fact, to eliminate having to write XSLT for aspects, and the accompanying boilerplate like

the `xsl:copy` elements and to generate them from a higher level description.)

As a second example, we can examine a case where around is helpful:

```
<xsl:template match="//filledTemplate[@name=$pipename]
         [@inside=$inside]//jpt:source[@point='pipe']">
static  FILE *afile;
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
#include &lt;unistd.h&gt;
int main()
. . .
```

In this case we are structurally bound by the standards of C coding which advocate placing variable definitions at the top of a file and having functions declared at file scope. This means we weave on the joinpoint that defines the source file of the Infopipe. The declaration of the variable occurs before the main code of the Infopipe, and the definition and declaration of the main function occur after. Since `main()` is not generated by default we add it using an aspect and then call the Infopipe startup code which subsequently spawns a thread to service our incoming Infopipes connection.

One of the interesting results of using XSLT and XML for this system is that aspects can introduce new joinpoints in the form of new XML tags. This means that one aspect can build upon an aspect that was woven into the code earlier (order of aspect weaving will be discussed shortly). In our example scenario, we insert timing code to measure how long various pieces of Infopipe code take to run the user function which can be used later in calculating CPU usage.

```
<xsl:template match="//filledTemplate
                     [@name=$pipename][@inside=$inside]//jpt:inport">
  <jpt:time-probe point="begin">
  // take timing here
  gettimeofday(&amp;inport_<xsl:value-of select="@point"/>_begin,NULL);
  </jpt:time-probe>
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
  <jpt:time-probe point="end">
  gettimeofday(&amp;inport_<xsl:value-of select="@point"/>_end,NULL);
  </jpt:time-probe>
</xsl:template>
```

The timing code is bracketed with XML that declares it, and the CPU monitoring code can then select it with a pointcut just like any other joinpoint:

```
<xsl:template match="//filledTemplate[@name=$pipename][@inside=$inside]
                    //jpt:inport[@point='callmiddle']
                    //jpt:time-probe[@point='end' ">
```

This brings us to the third part of the AXpect weaver – specifying the aspects to apply in the XIP specification. This is a very simple process in which we add `<apply-aspect>` statements to the pipe descriptions:

```
      <pipe class="vidSink" lang="C">
        <apply-aspect name="rate_controller.xsl" targetPct="20">
```

```
    <apply-aspect name="control_receiver.xsl" target="ppmIn"/>
    <apply-aspect name="cpumon.xsl" target="ppmIn">
      <apply-aspect name="timing.xsl"/>
      <apply-aspect name="sla_receiver.xsl" doc="uav.xml"/>
    </apply-aspect>
  </apply-aspect>
  <ports>
    <inport name="ppmIn" type="ppm"/>
  </ports>
</pipe>
```

Note that we can nest the `apply-aspect` elements to declare dependencies of one aspect upon another. Since we invoke the XSLT processor multiple times, and neither the XSLT standard nor Xalan-C supports self-invocation, the evaluation of these statements is handled in a C++ program using Xerces-C, which is the platform the ISG is built around. The weaver proceeds recursively through the following steps on each pipe:

1. Retrieves the first `<apply-aspect>` element from the pipe 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. 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.
5. Once all aspects are applied, the entire XML result document is passed to the last stage of the generator.

This algorithm implementation only required an additional 79 lines of C++ code be added to the generator application. The bulk of the weaver complexity is contained by the XSLT weaver.

5 Our Sample Application

We used the AXpect weaver and Infopipes to implement the sample application which we described earlier in the paper. We now discuss the implementation of aspects to fulfill the QoS requirements of the rate-adaptive image-streaming application.

The timing aspect hooks on to all join points that designate an executable block of code. This can be done in an efficient fashion by using the pattern matching to select entire sets of joinpoints around which to install timing code around. Complementing this is creating new variables to hold the timing measurements which we do by creating their names at aspect-weaving time.

On top of this we install the CPU monitoring code. This code installs around the join points for timing, specifically the timing points that designate the call to the middle-method code. Instead of using start-to-end elapsed time which would only provide a measure of how long it took to execute a joinpoint, we measure end-to-end so that we have a measure of the total time for the application to complete one "round-trip" back to that point. We can compare this to the system-reported CPU time to calculate the percentage of CPU used by this process.

The control channel sends data between the two ends of the Infopipe. We used a socket independent of the normal Infopipe data socket both to avoid the overhead of demultiplexing control information and application data and to piggyback this functionality on top of the OS demultiplexing which would be performed, anyway. Also, separating these two flows of information should improve the general robustness of the application as there is no possibility of errant application data being interpreted as control data or of misleading data being injected as control data somehow.

Finally, there is the SLA aspect. During weaving, it reads an external SLA document which specifies the metrics and tolerances of the values the SLA needs to observe and report. At run time, the SLA reads the CPU usage values and sends them through the control channel to the video; once received, the SLA acts based on the returned value. In our example, the SLA can set a variable to control if and for how long the sender enters `usleep()` to adjust its rate control.

We compiled the sample application and ran it with a "strong" sender, a dual 866MHz Pentium III machine and a "weak," resource-constrained receiver, a Pentium II 400MHz. Running without any controls on resource usage, the video sender is able to capture roughly 36% of the receiver's CPU. Using the CPU control, we are able to bring the CPU usage back to a target 20±5% range.

We have observed so far that our aspect files are generally larger than they amount of code they actually generate. However, this tradeoff is appropriate considering the increase in locality of code and reusability (some of these aspects, such as timing and CPU monitoring, have been reused already in another demo). In fact, when we examine the files altered or created by the aspects in this application, we see that an aspect such as the sender-side SLA code can alter four of the generated files and then add two more files of its own. In all, the QoS-aware application is 434 lines longer than the base application that is not QoS aware. Without support from a weaver to help manage code, it would obviously be more difficult to keep track of these 434 lines if they are handwritten into the base set of 18 files versus the six AXpect files.

(See also http://www.cc.gatech.edu/projects/infosphere/online_demos/WeaveDemo)

6 Related Work

The AOP and code generation community is actively exploring the new possibilities in combining the two including SourceWeave.NET [8], Meta-AspectJ[9], two-level weaving

[10], and Xaspects [11].

Before that, The AOP community has worked diligently on weavers for general purpose languages such as Java and C++. This has resulted in tools such as AspectJ, AspectWerkz, JBossAOP, and AspectC[4,13,14,15]. Generally, development of weavers for these platforms requires continual and concerted effort over a fairly long period of time. Other work has tackled separation of concerns for Java through language extensions, such as the explicit programming approach of ELIDE project [16].

DSLs have also often been implemented on top of weavers. Notable in this area is the QuO project, which uses a DSL from which it generates CORBA objects which are called during runtime to be executed at the join point to implement quality of service. However, the QuO project does not weave source code. Instead, it alters the execution path of the application therefore imposes invocation overhead [17]. Bossa uses AOP ideas to abstract scheduling points in OS kernels, but again does not do source weaving; each joinpoint triggers an event and advice registers at events in order to run [18]. Because of the use of aspects in conjunction with DSLs, the XAspects project is studying the use of aspects to implement families of DSLs. Still, this project uses AspectJ as the source weaver and therefore focuses only on Java as the target language [11]. The Meta-AspectJ package also targets enhancing the power of code generators and using code generation plus AOP to reduce complexities for implementing security and persistence [9]. Work has been done using XML in the AOP arena; however, this work has concentrated on using XML to denote the abstract syntax tree [18]; conversely, it has been used as the aspect language syntax as in SourceWeave.NET to weave aspect code in the bytecode of the .NET the Common Language Runtime (CLR) [8].

7 Conclusion and Ongoing Work

We have shown that even adding a relatively simple QoS requirement can entail widespread changes to an application and that those changes can be spread throughout the entire application. To address this, we described the AXpect weaver. The AXpect weaver can use information from a WSLA and integrate new code into source code generated from an Infopipes XML specification. Our target application used the weaver to add new functionality to a C program which realized an image-streaming with responsiveness to CPU usage constraints on the sender end of the image stream. For future work, we are continuing to explore the space of applications for weaving, and we have already demonstrated early application of the weaver to C++ programs with further plans for Java. Also, we are investigating Infopipes support for Web Service applications.

8 Acknowledgements

The authors are grateful for the input of Charles Consel (INRIA, University of Bordeaux, France); Ling Liu, Younggyun Koh, Wenchang Yan, and Sanjay Kumar (Georgia Institute of Technology, Atlanta, GA), and Koichi Moriyama (SONY Corp., Japan); Part of this

work was done under DARPA funding.

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. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin. "Aspect-Oriented Programming." *Proceedings of the 15th European Conference of Object-Oriented Programming* (ECOOP 2001). June 2001.
4. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. "An Overview of AspectJ." *Proceedings of the European Conference of Object-Oriented Programming* (ECOOP 1997). June 1997.
5. C. Pu, Galen Swint, C. Consel, Y. Koh, L. Liu, K. Moriyama, J. Walpole, W. Yan. Implementing Infopipes: The SIP/XIP Experiment, Technical Report GT-CC-02-31, College of Computing, Georgia Institute of Technology, May 2002.
6. 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. 2004.
7. M. Papazoglou. "Service-Oriented Computing: Concepts, Characteristics, and Directions." Fourth International Conference on Web Information Systems Engineering (WISE'03). December 2003.
8. A. Jackson, S. Clarke. "SourceWeave.NET:Cross-Language Aspect-Oriented Programming." Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE), Vancouver, Canada, October 24-28 2004.
9. D. Zook, S. S. Huan, Y. Smaragdakis. "Generating AspectJ Programs with Meta-AspectJ." Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE), Vancouver, Canada, October 24-28 2004.
10. J. Gray, J. Sztipanovits, D. Schmidt, T. Bapty, S. Neema, and A. Gokhale, "Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis." *Aspect-Oriented Software Development*. Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke, eds. Addison-Wesley, 2004.
11. M. Shonle, K. Lieberherr, and A. Shah. Xaspect: An Extensible System for Domain Specific Aspect Languages. OOPSLA 2003. October 2003.
12. S. Sarkar, "Model Driven Programming Using XSLT: An Approach to Rapid Development of Domain-Specific Program Generators," www.XML-JOURNAL.com. August 2002.
13. J. Bonér, A. Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org/.
14. JBoss. http://www.jboss.org/products/aop.
15. 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.
16. 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.
17. 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. May 28-30, 1998.
18. 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.
19. 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.