Clearwater: Extensible, Flexible, Modular Code Generation

Galen Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, Koichi Moriyama

Clearwater: Extensible, Flexible, Modular Code Generation

Galen S. Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu
CERCS, College of Computing
Georgia Institute of Technology
801 Atlantic Drive,
Atlanta, GA 30332-0280
swintgs@acm.org,
{calton, helcyon1}@cc.gatech.edu

Charles Consel
INRIA/LaBRI
Bordeaux, France
consel@labri.fr

Akhil Sahai
HP Laboratories, Palo Alto, CA
akhil.sahai@hp.com

Koichi Moriyama
Sony Corp., Tokyo, Japan

ABSTRACT
Distributed applications typically interact with a number of heterogeneous and autonomous components that evolve independently. Methodical development of such applications can benefit from approaches based on domain-specific languages (DSLs). However, the evolution and customization of heterogeneous components introduces significant challenges to accommodating the syntax and semantics of a DSL in addition to the heterogeneous platforms on which they must run. In this paper, we address the challenge of implementing code generators for two such DSLs that are flexible (resilient to changes in generators or input formats), extensible (able to support multiple output targets and multiple input variants), and modular (generated code can be rewritten). Our approach, Clearwater, leverages XML and XSLT standards: XML supports extensibility and mutability for in-progress specification formats, and XSLT provides flexibility and extensibility for multiple target languages. Modularity arises from using XML meta-tags in the code generator itself, which supports controlled addition, subtraction, or replacement to the generated code via XML-weaving. We discuss the use of our approach and show its advantages in two non-trivial code generators: the Infopipe Stub Generator (ISG) to support distributed flow applications, and the Automated Composable Code Translator to support automated distributed application deployment. As an example, the ISG accepts as input an XML description and generates output for C, C++, or Java using a number of communications platforms such as sockets and publish-subscribe.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *languages (e.g., description, interconnection, definition), domain-specific architectures*

General Terms
Languages

Keywords
Clearwater, Infopipes, AXpect, ISG, code generation, DSL

1. INTRODUCTION
Automating the generation of code for distributed systems software has been an established technique since the introduction of RPC stub generator [4]. However, significant research challenges remain for generating flexible, reusable, and modular distributed systems software. For example, environmental and design changes pressure the input language to change and evolve. Often, irrefutable forces external to a project such as mergers, acquisitions, or standards adoption dictate this evolution. Similarly, the generated code (output) often needs customization to a range of software and hardware platforms, also typically due to unyielding market and technology evolution. This constant evolutionary pressure of input and output formats has so far limited the practical life span of code generation tools developed for distributed system software.

Two of our recent research projects have encountered the issue of accommodating heterogeneous distributed system elements in code generation. In the first, the Infosphere project, our obstacle was encapsulating middleware for distributed information flow systems, which are characterized by continuous volumes of information traversing a directed workflow network [5][19]. The second project addressed the resource deployment problem whereby distributed applications should start efficiently and in provably correct order by simultaneously enforcing serialization constraints and leveraging the distributed system's inherent parallelism. In both cases, our challenge was building a generator for mapping evolving domain-level languages to multiple execution platforms (lower-level output languages). The result of our experiences was the Clearwater approach which applies XML, XSLT, and XPath to address these code generation challenges [6][8]. Our earlier publications addressed the contributions of the tools we developed. The contribution of this paper is to illustrate the practical and research advantages of using the Clearwater approach to code generation for domain-specific languages (DSLs) and present two generators built using the approach, ISG (the Infopipe Stub Generator) and ACCT (the Automated Composable Code Toolkit).

We can generalize the generator requirements needed to support ongoing research into the need for *extensibility*, *flexibility*, and *modularity*. Our reasons for each of these:

extensible — Extensibility is supported at two levels: for the domain and for the target implementations. In the Clearwater

context, domain extensibility means that new domain features can be encoded in the XML specification with minimal impact on pre-existing specifications. Furthermore, we want to support a variety of domain-level input sources (text files, program toolkits, GUIs, etc.). With regard to target implementations, extensibility addresses the problem of heterogeneity, a hallmark of complex distributed systems. Therefore, we required support for multiple general purpose languages and multiple communication layers as simultaneous output.

flexible — Our specification formats are ongoing research. So, the generators should be robust to changes in input specification, *i.e.* specification changes should require no or minimal re-writes to the generator. Likewise, supporting new implementation-level features and re-factoring of the generator code generally should not demand re-writing of domain-level elements or re-structuring of the intermediate representation.

modular — A developer frequently needs to make controlled changes to the generated code. For instance, quality of service often demands such consideration. These changes may be specific to the application for which we are generating code and therefore not suitable for general inclusion in the code generator. Supporting modularity encourages the writing of re-usable modifications for the generated code.

Traditional code generation techniques rely on developing a language and grammar, parsing inputs into a token stream, building a custom abstract syntax tree (AST), and then tailoring a code generator to the AST to produce output code. Consequently, a change to or extension of the specification language requires multiple simultaneous activities: creating the new domain language features, defining their lexical patterns, defining their grammar rules, updating the AST design, and finally, reconciling the generator to the new AST. Only when the developer has completed all these can he or she construct a demonstration application and test the new produced code – a non-trivial task on its own. If multiple targets are required, the developer must change and test the generator for each and every target (implementation) platform. This overhead proscribes specification flexibility or extensibility since it magnifies even small changes. Code modularity is not readily addressed in any platform independent fashion, either.

By using XML and XSLT, we can sidestep or mitigate these dependencies and support cross-language development and multi-input format specification while maintaining extensibility in terms of language support and code generation features. XML provides an extensible and modular specification format for the intermediate representation and the AST; and XSLT, with its use of XPath, offers flexible structure-independent access to the information in the AST. Interestingly, by using XML meta-information within the generator itself and then weaving in new code after generation, we can also achieve our goal of modular generated code.

Our project parallels several others using XML and XSLT for code generation. For example, the SoftArch/MTE and Argo/MTE teams have also had favorable experiences using XML + XSLT generators to "glue" off-the-shelf applications together [7][13], and XML+XSLT is advocated for code generation tasks in industry as well [24]. To our knowledge, these efforts have not explored the issues of extensibility, flexibility, or modularity presented here. Although Karsai discusses a number of possible shortcomings in using XSLT+XML in a semantic translator [14], we have found the two technologies to be quite amenable as a core for code reuse through generation.

We have based two generators on this technique. The ISG underpins four types of input: Spi, a human readable format for Infopipes; Ptolemy II, a GUI builder for workflows; XIP, the XML description of Infopipes and native format for ISG; and WSLA, the Web Service Level Agreement specification. ACCT, which is less mature, supports CIM-MOF. For output, the ISG generates C, C++, and Makefiles, and ACCT generates Java and SmartFrog's specification language [21]. These experiences suggest that the Clearwater approach generally is not limited to any particular input or output language.

The rest of the paper is structured as follows: First, we introduce our target application domains. Following that, we present a general overview of our DSL compilation process. Then, we discuss how XML and XSLT in the Clearwater approach introduce the extensibility, flexibility, and to code generation. Third, Section 4 presents the ISG code generator, its AXpect weaver module, and ACCT to illustrate their operation and how our goals of extensibility, flexibility, and modularity are borne out in those systems. Next, we discuss and present our application-building experiences using the generators with respect to code performance and functionality, and finally, we present related work and our conclusions.

2. APPLICATION BACKGROUND

The Clearwater approach was developed in the course of building the ISG for the Infosphere project. We refer the reader to [19] for detailed discussion, and will present enough information here to provide an illustrative context that demonstrates Clearwater benefits in practice and makes this paper self-contained. Our second application domain, for ACCT, will be described in Section 4.3.

A simple Infopipe instance has two ends – a consumer (inport) end and a producer (outport) end – and implements a unidirectional information flow from a single producer to a single consumer. Between the two ends is the developer-provided Infopipe middle, which processes or transforms information. In operation, an information producer exports and transmits an explicitly defined and typed information flow, which goes to a consumer Infopipe's inport. After appropriate transportation, storage, and processing, the information then flows to a second information consumer which may reside in a different geographic location.

The Infopipe abstraction is language and system independent; as a consequence, generated stub of code in the abstraction is able to hide the details of marshalling and unmarshalling parameters for languages, hardware, communication middleware, etc. There are three sources of problems in the implementation of a stub generator: (1) the heterogeneity of languages, operating systems and hardware, (2) the translation between the language level procedure call abstraction and the underlying communication library implementation, and (3) customization to a particular application's requirements.

3. CLEARWATER

We will first discuss a Clearwater generator's relation to traditional compiler architecture, and then we will present and discuss how XML and XSLT provide flexibility, extensibility, and modularity inside that model.

3.1 Overview

From an architectural viewpoint, Clearwater adopts the compiler approach of multiple serial transformation stages – a code generation pipeline. The Clearwater hallmarks are that stages typically operate on an XML document that is the intermediate representation, and XSLT performs code generation. The overall process:

1. Compile to intermediate format (High Level Language-to-XML). This is mainly a straightforward translation from a human-friendly representation into XML.
2. Pre-processing of the XML intermediate representation. We lookup extra information from disk, if needed, resolving names, etc., and add tie the new information into the XML intermediate representation.
3. Code generation via XSLT that transforms our representation from XML to XML+Source code. We preserve the specification and generate new source code into the (pre-processed) specification. In this phase, we may also generate additional XML tags along with the source code to be used in the next step. One might also consider this as a parse tree annotated with source code.
4. Post-processing. This step may involve iterative code generation steps that consume and produce XML elements.
5. Write generated source to files, directories (transform XML+Source to pure source code).

Stage two reads and parses an XML input file to produce a DOM (Document Object Model [16]) tree in memory, a decoupling that facilitates one generator's serving multiple high-level languages. In practice, we have kept the high-level compilers of stage one independent from steps 2 through 5 and use the XML intermediate format as the primary input for experimentation as this allows for greater flexibility in terms of research. However, we could easily opt to wrap step 1 and steps 2 - 5 via a shell script. Stage two also preps the intermediate language for processing by the code generator. Following that, stage three generates code via XSLT resulting in a new XML document containing both the specification and newly generated code. Stage four provides aspect weaving and modular modification of the generated code. Finally, stage five writes the files to disk by stripping their XML accoutrements.

3.2 XML: Extensible Domain Specification

XML's chief contribution to the Clearwater approach is that it introduces extensibility at the domain-language/domain-specification level. This stems from XML's simple, well-defined syntax requirements and ability to accept arbitrary new tags thereby bypassing the overhead encountered when managing both a grammar and code generator.

As an example of specification extension, consider a scenario in which a developer adds new information specific to a target architecture. In Infopipes, an example is that native sockets support only data transmission, but the ECho event middleware supports "safe", uploadable filters on events [12]. To accommodate the filter functionality at the domain level, the ECho developer must first extend the specification with new filter descriptions. Whereas the use of a grammar based approach encounters the difficulties listed in the introduction, in the Clearwater approach adding new elements to the specification document alongside existing elements requires no changes to the parser, lexer, syntax checker, or grammar definition.

In maintaining grammars, a developer spends a great deal of time explaining the structure of a domain language to the parser by defining tokens (lexing) and simultaneously determining what token orderings are valid. Deviations from defined rules break the lexer/parser and experimentation becomes difficult. Furthermore, most approaches to generation create an abstract syntax tree based explicitly on the grammar for the language. Therefore, any language change finds its way into the parser's AST, too, and from there the code generation logic that interacts with the AST must *also* be changed.

Because XML always represents a fully-parenthesized syntax tree, document structure is always explicit (through element nesting and angle brackets), and rules that govern the structure are (often) implicit. Consequently, a changed specification format very often can be accepted without syntactic complaints by the existing generator package. This extensibility sidesteps the problems of parsing by isolating them from the code-generator chain. Because XML documents implicitly encode production rules, developers of domain language generators benefit by avoiding the premature tying of the *generator* to a particular concrete grammar. Users can add new XML tags to a well-formed XML document, and therefore to their language grammar, provided the changes maintain well-formedness.

XML has several advantageous properties for being a general specification format. First, XML defines a very simple lexical pattern for characters that allows automatic tokenization by the XML document parser. Reserved words which create a "block" of code with some meaning are either 1) enclosed in angle brackets and given the meta-name "element" (*e.g.*, `<subpipes>` in Figure 1), or 2) form a quote-delimited name-value pair specific to an element and forms an "attribute" (*e.g.*, `name="UAV"`). New reserved words can be added to a language by adding new elements or attributes to the XML representation. XML itself only reserves two symbols, '<' and '&', the first to identify elements and the second as an escape character.

We exploited extensibility to great advantage during ISG development in that we were able to maintain multiple researchers' efforts simultaneously without concern for specification mismatches. As it turned out, each researcher created a slightly different code generator that operated from the same core XML document. For instance, one developer worked on support for aspects (AXpect) and introduced tags to support that effort while another developer worked on mobile data filters with his own custom tags added to the core document. Importantly, the developers could re-use the documents of each other for various testing purposes without worrying about breaking their own code.

Concluding our XML discussion, one last useful feature, though not strictly germane to fulfilling extensibility, is the XML namespace. An XML namespace, in principle, performs for XML elements the same function as a namespace in a general language, partitioning meaningful tokens into non-colliding subgroups. In practice, this means that several overlapping trees of information can exist in the same document. Clearwater uses namespaces to clarify modularity for XML-weaving.

3.3 XSLT: Flexible, Extensible Generation

In addition to the extensible specification, we needed an extensible and flexible code generator that operated from the specification. By *flexible*, we mean that the code generator is tolerant of

Specification 1

```
<datatype name="FloatArray">
  <arg name="SIZE" type="integer"/>
  <arg name="buff" type="string"/>
</datatype>
<pipe name="UAV">
  <subpipes>
    <subpipe name="Sender" pipeOf="Sender"/>
    <subpipe name="Receiver" pipeOf="Receiver"/>
  </subpipes>
  <connections>
    <connection comm="ECho">
      <from pipe="Sender" port="out1"/>
      <to pipe="Receiver" port="in1"/>
    </connection>
  </connections>
</pipe>
```

Specification 2 - Extended

```
<datatype name="FloatArray">
  <arg name="SIZE" type="integer"/>
  <arg name="buff" type="string"/>
</datatype>
<filter name="GREY">
  <in type="ByteArray"/>
  <out type="ByteArray"/>
</filter>
<pipe name="UAV">
    <subpipes>
      <subpipe name="Sender" pipeOf="Sender"/>
      <subpipe name="Receiver" pipeOf="Receiver"/>
    </subpipes>
    <connections>
      <connection comm="ECho">
        <from pipe="Sender" port="out1"/>
        <to pipe="Receiver" port="in1"/>
        <use-filters>
          <use-filter name="GREY"/>
        </use-filters>
      </connection>
    </connections>
  </pipe>
```

Figure 1. Specification 1 is a fragment from a basic Infopipe specification. We can extend our specification, without modifying any grammars and using the same parser, to include the '`filter`' construct and '`use-filter`' modifier as in Specification 2.

changes to the AST. By *extensible*, we mean similarly to extensible specification, new target outputs or functionality can be added to the generator. The Clearwater approach fulfills both of these requirements by using XSLT to generate target code. First, we will describe XSLT and its co-standard XPath [9]; then, we will address flexibility; and finally, we will discuss extensibility to new outputs.

XSLT, the Extensible Stylesheets Language for Transformations, is a (Turing complete) language for converting XML documents into other types of documents – typically another XML or HTML document. Each XSLT script, or stylesheet, is a collection of templates, and in the Clearwater approach, each of these roughly corresponds to some unit of transformation from specification to generated code. Practically, the flexibility requirement means that XSLT generator code must have the ability to ignore unknown tags and still generate correct code that implements a portion of the specification. It is the use of XPath that infuses XSLT with its flexibility; XPath allows a developer to refer to locations and groups of locations in an XML tree similar (syntactically) to how a hierarchical file system allows path specification. It has two important features improving beyond basic file paths, however.

First, XPath has a '//' ("descendant-or-self") 'axis' that encourages writing structure-shy paths [17]. A structure-shy path is one that is not closely tied to the absolute ordering and nesting of nodes in a tree. The '//' and the structure-shy qualities of XPath allow a developer to perform references to information without regard to explicit placement. Second, XPath provides predicate execution. Because structure-shy paths do not necessarily indicate a single, unique XML element, it may return a set of nodes from the parsed document. Predicates can narrow these nodesets to small or singleton subsets. In Figure 2, we illustrate moving data-descriptions within the document does not break a properly written XPath statement that retrieves that data from a datatype declaration located in various places within the specification document.

In operation, a language developer can write a template to be activated in one of two fashions. First, the template may be invoked explicitly by name – this is just as one calls a procedure or function in other languages. Second, the template may be invoked implicitly by an XPath pattern match. In pattern matching, developers use XPath to select groups of elements (nodesets) from the source XML document. These are matched to patterns specified per template, and when an appropriate match is selected using XSLT `apply-templates` instruction, a template will execute.

As an example of template execution, consider the ISG code generator's operation over a XIP document. A XIP document can be represented as a tree with a single element 'xip', containing sub-elements. The 'pipe' sub-element encapsulates the data that describes an Infopipe. For a C generation template, the pattern

XPath: `//datatype[@name='ppmType']/arg[@type='long']`

```
<datatype name="ppmType">
  <arrayArg name="mag"
     type="char" size="2"/>
  <arg name="width" type="long"/>
  <arg name="height" type="long"/>
  <arg name="maxval" type="long"/>
  <arg name="pictureSize" type="integer"/>
  <arrayArg name="picture"
     type="byte" size="pictureSize"/>
</datatype>

<pipe  lang="CPP" class="ReceivingPipe">
  <apply-aspect name="receiver_gpce.xsl"/>
   <ports>
     <inport name="in" type="ppmType"/>
  </ports>
</pipe>
```

```
<pipe  lang="CPP" class="ReceivingPipe">
    <apply-aspect name="receiver_gpce.xsl"/>
   <ports>
     <inport name="in" type="ppmType">
       <datatype name="ppmType">
         <arrayArg name="mag"
            type="char" size="2"/>
         <arg name="width" type="long"/>
         <arg name="height" type="long"/>
         <arg name="maxval" type="long"/>
         <arg name="pictureSize"
            type="integer"/>
         <arrayArg name="picture"
            type="byte" size="pictureSize"/>
       </datatype>
  </inport>
  </ports>
</pipe>
```

```
<pipe  lang="CPP" class="ReceivingPipe">
  <datatype name="ppmType">
    <arrayArg name="mag"
       type="char" size="2"/>
    <arg name="width" type="long"/>
    <arg name="height" type="long"/>
    <arg name="maxval" type="long"/>
    <arg name="pictureSize" type="integer"/>
    <arrayArg name="picture" type="byte"
         size="pictureSize"/>
  </datatype>
  <apply-aspect name="receiver_gpce.xsl"/>
   <ports>
     <inport name="in" type="ppmType"/>
  </ports>
</pipe>
```

Figure 2. In this simple example, the XPath expression returns all the data members of type '`long`' for the type named '`ppmType`' equally well in all three cases even though datatype has been moved within the specification document – first, as global information, then as a localized association with a pipe, and finally as an association with a single port on a pipe. Of course, these changes do not affect XML parsing either. Such an XPath expression is used in code generation, for instance, when generating datatypes containers (*e.g.*, a `struct` or `class`) or marshalling code.

"`/xip//pipe[lang='C']`" will execute for the subset of Infopipes with a chosen output language of "C" (`<pipe lang="C">` in the specification) when the `apply-templates` selects "`/xip//pipe`" – which comprises all Infopipe specifications regardless of implementation language specified. If we also had a template for C++ that matched "`/xip//pipe[lang='CPP']`", then the same `apply-templates` command would cause them to be executed, too. On the other hand, if there is no match then that section of the specification will be ignored without breaking the generator. For example, the specification states "`pipe lang="java"`," but there is no "`/xip//pipe[lang='java']`" to recognize it.

Extensibility in the Clearwater approach emerges when runtime compilation, pattern matching, and stylesheet importation combine. In the ISG, language-specific XSLT files are imported into a single `masterTemplate.xsl` file, and pattern selection from the specification controls the execution. We re-apply the approach at the communication layer level in our generator thereby establishing extensibility for various communications packages.

The first enabler of extension is XSLT's option to use either call-by-name or pattern matching. The effect of having both semantics is that it is possible to alternate control of the generation process between the generator and the specification. For example, using a pattern to match the C Infopipes, as above, lets the specification control entry into that group of templates. These templates may call by name other templates that automatically generate header files and make files – at which time the generator-code controls the code production. In our experience with ISG, it is quite common for us to use both. Often, we create call-by-name templates to separate code generation into smaller fragments when a lot of code is to be executed for a single pattern match.

Second, XSLT also supports importation of stylesheets, as shown in Figure 3, so that complex stylesheet behavior can be composed from multiple simpler stylesheets. Alternatively, a complex stylesheet can be broken into smaller stylesheets for better organization. As an example of this technique, in the ISG we use separate stylesheets for our C and C++ generation and further deconstruct those into smaller stylesheets based on the communication mechanism supported (e.g. TCP or the ECho middleware package).

Finally, XSLT is runtime compiled allowing output to change easily and quickly. One might mimic this functionality through external resource strings if developing in a compiled, object-oriented environment like Java, but generator development then becomes limited to variations on pre-identified strings. Consequently, any reorganization that does not already fit the established mapping from high-level language to the implementation language will require changes to a generator object. XSLT allows easy change of the output without re-writing objects or re-compiling. This shortens the development cycle and also lowers the maintenance hurdle.

```
masterTemplate.xsl
<xsl:import href="allMake.xsl"/>
<xsl:import href="CPP/CPP.xsl"/>
<xsl:import href="C/C.xsl"/>
...
<xsl:apply-templates select="/xip//pipe"/>

C.xsl
...
<xsl:template match="/xip//pipe[@lang='C']">
...

CPP.xsl
...
<xsl:template match="/xip//pipe[@lang='CPP']">
...
```

Figure 3. By inserting an `import` directive and using XPath pattern selection for the target language, extension to new output targets is easy and independent.

3.4 XML+XSLT: Modularity and Weaving

Finally, one sizable advantage the Clearwater approach has leveraged is the fact that every XSLT document is valid XML. Consequently, using the Clearwater approach one can embed new XML tags in code-generating XSLT but affect neither speed nor correctness of the transformation process. Then, when this XSLT generates output code, these XML tags are replicated the tags into the target code where they act as semantic markers to expose the domain structure of the generated code. Each block of generated code becomes a module that can be replaced or augmented. These blocks support aspect-weaving for the generated document. In the ISG, the weaving capability is implemented by the AXpect weaver which we discuss in detail in Section 4.2.

XML, XSLT, and XPath combine to make the mechanics of these code substitutions and additions easy. Given a generated document with the aforementioned XML tags, an XSLT template can use XPath to find those tags and replace or augment the existing code with new code and tags. From an AOP vantage point, XPath selects pointcuts and XSLT encapsulates advice over the joinpoints. The XSLT processor performs the task of joinpoint identification and weaving for us. Note that the only language dependency in this process is the direct dependency between the advice and the target source language so that language-specific weavers are bypassed. We have executed our AXpect weaver on both C and C++ Infopipes.

Consider the excerpts in Figure 4. The `jpt:pipe` tags in the generator template denote the code that performs shutdown tasks for an Infopipe which consists of successively shutting down inports and outports. On the right, we can see that the tags are kept with the code after generation and clearly label the purpose of that block of C code. From an AOP perspective, these tags form a set of joinpoints on the underlying generated code. Each joinpoint maps some logical domain feature into the "physical" implementation in a target language. There are two major benefits from this. First, it allows code generation to be modular. If we need to replace some default generated functionality, we can. For instance, Infopipe communicate connection information via files over NFS, but we replace that code with hard-coded connection information when we experiment in emulated distributed environments. Second, it allows us to insert features into the generated code that are otherwise orthogonal to the domain language. A good example of an orthogonal feature encapsulation is a WSLA governing Infopipe performance [26].

4. IMPLEMENTATIONS

Using the Clearwater approach, we have implemented two code generators. ISG drove the development of the approach; it converts Infopipe specifications, XIP, into general purpose language implementations and supports AOP via its AXpect module. The second generator, ACCT, resulted from a joint venture with HP

Generator Template

```
// shutdown all our connections
int infopipe_<xsl:value-of select="$thisPipeName"/>_shutdown()
{
  <jpt:pipe point="shutdown">
  // shutdown incoming ports <xsl:for-each select="./ports/inport">
  infopipe_<xsl:value-of select="@name"/>_shutdown(); </xsl:for-each>
  // shutdown outgoing ports <xsl:for-each select="./ports/outport">
  infopipe_<xsl:value-of select="@name"/>_shutdown(); </xsl:for-each>
  </jpt:pipe>

  return 0;
}
```

Emitted XML+Code

```
// shutdown all our connections
int infopipe_sender_shutdown()
{
  <jpt:pipe point="shutdown">
  // shutdown incoming ports

  // shutdown outgoing ports
  infopipe_ppmOut_shutdown();
  </jpt:pipe>
  return 0;
}
```

Figure 4. "Generator Template" displays the XML markup in the XSLT that generates shutdown code for an Infopipe – calling shutdown functions on inports and outports. Emitted code shows how this markup persists after generation and denotes, in this case, the shutdown of the Infopipe's lone outport.

Labs. Though it is newer and less developed, it is still built upon the XML+XSLT approach of Clearwater.

4.1 The ISG Generator

The current version of the ISG generator is a hybrid language application of C++, providing the XML parser and DOM document interface, with an embedded XSLT processor. We recognized the need for a general purpose language upon discovering two limitations of pure XSLT. First, file support is limited, and while new standards are enabling multi-document output, this was not true at the time we first wrote the ISG. Second, XSLT has only recently added the capability of accessing created XML document fragments at run-time. This limited the ability to construct XML fragments with information from a document in any sort of recursive fashion. Because of this, we use the C++ and an XML package to perform pre-generation processing, which involves resolving connections between Infopipes and retrieving specifications from the repository. This process specifically involves recursively descending through Infopipe descriptions and retrieving multiple documents from disk from the repository which were then melded together to form what we call the XIP+ document actually used for generation.

As we mentioned one goal was to support multiple communication layers and implementation languages simultaneously. In the ISG, C and C++ can be created concurrently from a single specification. For example, a C Infopipe may communicate via ECho event channels to a second C Infopipe, which in turn sends data over a TCP connection to a C++ Infopipe. Even supporting several output options, the code generator is a fairly manageable in terms of overall size (see Table 1). In addition to the targets listed in the table, we also have varying support for additional language and communication layer pairings with the ISG. These include C++ using CORBA, local IPC, or local function calls, and Java and XML over TCP. The XSLT templates and XML AST encourages language dependencies to be isolated from the language independent code of which is tailored domain-level Infopipes information.

Mirroring our multi-output goal, we support multiple inputs via multiple high-level language converters. Spi (Specifying Infopipes) is a human-friendly language which is compiled through the Ply parser/lexer package for Python into XIP. As a second high-level Infopipes tool, we augmented the Ptolemy II toolkit to support Infopipes. The XML based Ptolemy II representations are transformed via XSLT into XIP which can then be executed by the code generator.

Figure 5 illustrates the stages of the ISG and AXpect weaver, which we will describe more fully in the next section, and Table 1 provides corresponding source sizes (calculated by David A. Wheeler's SLOCCount). During generation, the specification AST is maintained as a DOM tree in-memory. Leaving discussion of the AXpect weaver for later, ISG code generation proceeds as follows:

1. The Infopipe XIP description is divided into several sections of datatypes, pipes, filters, etc. and writes the specification fragments to the repository.
2. Elements designating which pipes to build are retrieved from the input XIP. Each forms the nucleus of a new document, which we term XIP+, which is built stored specifications and has verbose connection information.
3. The ISG passes the document to and invokes an XSLT processor to execute generation templates. Both the generated code and the reconstituted XIP+ are retained after code generation.
4. The specification+code is passed to the weaver (described in the next section).
5. Finally, XML markup is removed, and the code is deposited into files and directories, ready for use in an application.

The XSLT templates encapsulate the language and output dependent components of code generation. Figure 6 illustrates the organization of XSLT templates, and Table 2 presents their sizes.

Table 1. Lines of C++ code in language *independent* ISG modules excluding external libraries (e.g. XSLT processor). This code is not in the templates and essentially performs management of the generation process

Code (generation stage)	*Line Count*
Pre-process (1, 2)	756
Generation (3, excl. XSLT)	40
Weaver (4)	90
Write Files (5)	469
Shared all stages	134
Total	**1489**

Table 2. Lines of code (XSLT and target language) in XSLT templates that constitute the language *dependent* modules of code generation

Code		*Line Count*
master (Makefiles)		56
C	core	276
	TCP	679
	ECho	437
C++	core	515
	TCP	612
C/C++ shared		211
Total XSLT		**2773**

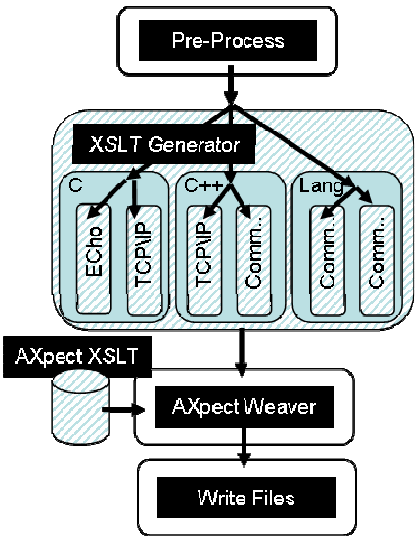

Figure 5. The ISG. The shaded and crosshatched areas are the only output-language dependent modules of the generator. The result of the "XSLT Generator" stage is a single containing generated all code and the specification.

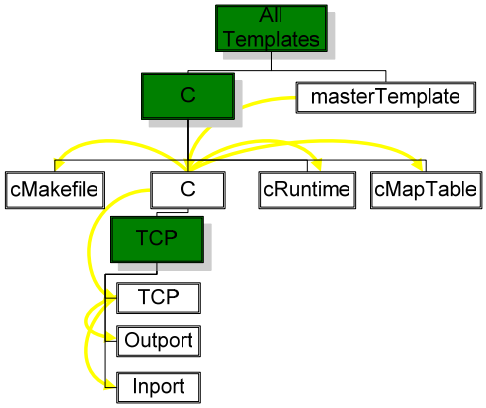

Figure 6. XSLT template organization for C/TCP Infopipes. Shaded boxes are directories, clear boxes are XSLT files, and arrows represent XSLT inclusion.

Table 1 and Table 2 provide some interesting information with regards to the ISG and the Clearwater approach. First, we note that while it is not a large application, it is of significant size comprising well over 3000 lines of code. Second, we can observe that XSLT allows significant factoring of common code between the differing communication layers for the C templates and even between C and C++. Were this code (C core and C/C++ shared) not factorable, *i.e.* were it required in both the TCP and ECho generators, it would increase them by over 70% and 110% respectively.

At the top of the hierarchy, the ISG invokes a master template located in a well-known directory that run-time includes templates for each supported language. Each of these language templates resides in a directory dedicated to XSLT templates that support that code generation. In the figure, we see that the C subdirectory has templates for generation of C core code, C runtime support, and a map table for mapping Infopipe specific data primitives to C types. Within the C subdirectory there is a TCP subdirectory that contains the XSLT templates for implementing TCP connections between Infopipes. Likewise, ECho for C has a parallel subdirectory and allows the two communications implementations to share the core code. Likewise, there is a CPP (C++) subdirectory for our C++ generation templates with it, too, having multiple communication language subdirectories.

Our C implementation follows the traditional approach of dividing code into file-level modules and each file corresponds to one functional unit of an Infopipe. The generated C++ implementation follows an object-oriented decomposition into base classes and subclasses corresponding to functional units. Despite this, the two implementations can have shared code. For instance, C++ directly generates using C runtime support templates code for publishing and discovering Infopipe connection information. Other times the generators have structural similarities, such as in unmarshalling code, but due to language idioms are not shared, *e.g.*, unmarshalling data to a `struct` for C but a `class` for C++.

4.2 The AXpect Module

One of the most important goals of the Infosphere project is addressing quality of service, such as data latency, security, or resource control, for information flow systems. However, our basic code generation did not include generation of any code to support QoS. Furthermore, it seemed that if we did add quality of service it would be difficult to anticipate all possible QoS scenarios. In light of this, we decided an aspect-oriented approach to QoS was warranted. Unfortunately, while there are several Java aspect weavers, there are no successful weavers for C or C++, our primary target languages. Still, some projects had been successful at marrying DSL techniques and AOP [3]. Our efforts in this space produced the AXpect weaver. (See also [20] and [26] for more details about AXpect; [15] for more about AOP in general.)

AXpect weaving occurs subsequent to code generation and prior to file output. Its implementation has three parts. First, we tag the code generation templates with new XML that demarcates Infopipe operations in the generated code. These *joinpoints* have two functions. First, they map the domain data contained in the Infopipe specification into the generated code, and second, they expose language-level features such as classes or header files which might otherwise be "hidden" during code generation. The second part of its implementation is that we can write an XSLT template that encapsulates an aspect. In this aspect, *pointcuts* are expressed as XPath statements selecting the XML joinpoints. Finally, the third piece is a C++ wrapper and declarative `<apply-aspect>` tags in the XIP to integrate the weaving process into the ISG, manipulate files, and resolve dependencies; the weaving algorithm recursively executes as follows [26]:

1. Retrieve the first `apply-aspect` from the specification.
2. If the aspect depends on more aspects, 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 based on the appropriate output-language for the target pipe.
4. The weaver code then passes the aspect and the XIP+ generation document to the XSLT processor. The result is a new XIP+ document which contains the specification, generated code, new woven code, and joinpoints.
5. The resultant XIP+ document (still a DOM tree in memory) serves as new 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

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. 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.

We have found AXpect to be useful in controlling QoS and implementing web service level agreements [26], and that the approach encourages good reuse of QoS code [25].

It is interesting to note that the exact same framework we use for implementing basic Infopipes and their functionality can be used to implement support for part of the WSLA specification without modification. All that is required is to insert new XSLT templates that implement the desired WSLA functionality as aspects. Furthermore, this functionality can be developed on an as-needed basis since we can choose which aspects to implement at which times and since XSLT templates can call other XSLT templates to form libraries of WSLA code generation functions.

4.3 The ACCT Generator

Our second generator, ACCT, we developed in conjunction with HP Labs. ACCT connects the design stage to the deployment stage in a business-objective driven closed loop management system for utility computing environments [23]. ACCT maps high-level constraints on distributed application deployment, such as start-up sequencing, into a low-level deployment plan. Combining such tools moves application deployment from the realm of brittle, uncertain, *ad hoc* scripts to provably correct and efficient automation. Cauldron, a high-level reasoning engine [22], produces a deployment plan for a distributed application, and SmartFrog provides deployment management daemons that can execute deployment workflows. We had two important problems: first, perform the non-trivial mapping of Cauldron's MOF into SmartFrog's requirement for Java source and SmartFrog workflow specification; second, accommodate tools beyond our basic set of Cauldron and SmartFrog.

ACCT shared similar goals to the ISG: 1) Translate Cauldron's high-level Managed Object Format (MOF) to low-level SmartFrog objects; 2) after initial support for Cauldron/SmartFrog, support translations in multiple deployment and resource management tools, and 3) support formal verification of deployment schemes. Given the early stages of this project, we have concentrated so far on the first two goals, but ACCT is still built using Clearwater's hallmark of XML for the specification and AST and XSLT for code generation.

There are two mismatches between Cauldron and SmartFrog. First, they have no common interchange specification. Cauldron emits MOF, but SmartFrog requires a SmartFrog workflow document plus a set of Java class definitions. Second, Cauldron generates a deployment plan consisting of pairwise dependencies between application components whereas SmartFrog needs a complete workflow specification of all dependencies in order. ACCT fulfills both requirements.

Instead of using a C++ harness like the ISG, ACCT's uses Java to manage generation, although the first version of ACCT was pure XSLT. ACCT is over 2000 lines of code has three major stages: 1) pre-processing to convert MOF to an XML format, 2) data extraction, and 3) translation to code. In the first stage of ACCT, it compiles the Cauldron-generated MOF input into CIM-XML, an XML formatted document for the Common Information Model. This is passed to stage two which has three XSLT generators that extract the proper data to generate source code.

SmartFrog itself requires three types of files. First, a workflow file is created by converting pairwise event dependencies emitted in Cauldron MOF into totally-ordered and properly synchronized events in for SmartFrog workflow language. The other two files SmartFrog needs are ACCT-generated component definitions written in Java. These are also converted from data contained in the MOF, and define generic component functionality, and corresponding instances of components that define the fully parameterized (needed at run-time) definitions of the components. These generated specifications are converted wrapped into a single XML format called XACCT (XML for ACCT) that should provide flexibility for other deployment tools in the future. Finally, one more XSLT template strips the XML and provides the ultimate conversion into SmartFrog-deployable sources.

5. EXPERIENCE

So far, ISG-based information flow systems have shown favorable performance results when compared to traditional RPC systems. Particularly, they are able to obtain better bandwidth in synthetic benchmarks [27][28]. This indicates that our Clearwater architecture poses no inherent limit on the generated code when compared to a traditional generation tool like `rpcgen`.

We have used the ISG to build two differing C/TCP-based Infopipes systems. The first system we concentrated on was a simple, two Infopipe image streaming system with quality of service. In this scenario, a streaming image server fed a lightweight client

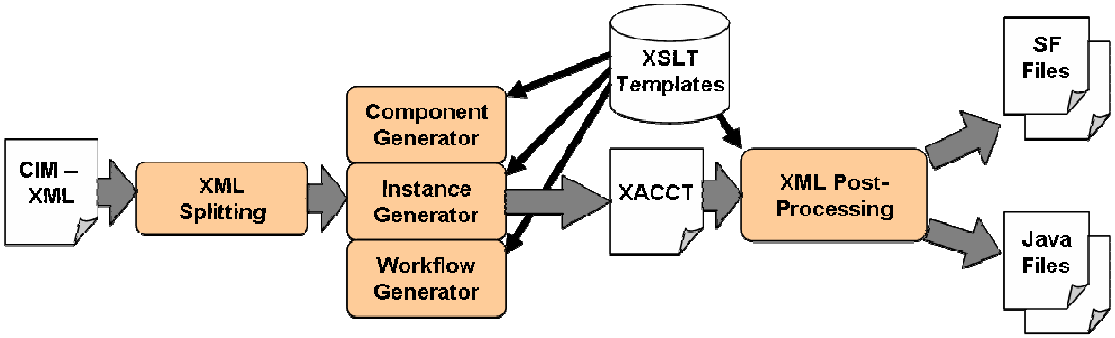

Figure 7. The ACCT generator maps CIM-XML into SmartFrog specification and Java code. ACCT splits a CIM-XML document into three parts as input to three XSLT-based code generators. After generation, ACCT comprises them into a single XACCT document which is stripped of XML by an XSLT then written to disk as output.

with limited CPU resource. We used the ISG to generate communication stubs and AXpect to add WSLA implementation code for the CPU monitoring and adaptation [26].

To implement the CPU monitoring and adaptation, we wrote six AXpect aspects (listed in Table 3) and a WSLA document that described the adaptation parameters. In our test case, we targeted 20% CPU usage for the receiver and adjusted our sender's rate based on returned CPU usage metrics. Even for such a simple application, the communication code generated by the ISG was nearly 1000 lines and over 400 more lines were added by aspects in the weaving process to implement CPU usage measuring, to install a control channel, and to add parameterization hooks from the application into the WSLA. In the end, about 30% of the generated application skeleton code was dedicated to providing QoS measurement and adaptation. Most significantly, the use of the AXpect weaver allowed this additional code to be encapsulated for later re-use rather than being "one-off" modifications applied for each application.

Secondly, we have used the ISG and AXpect weaver to build a variant of Linear Road benchmark [25]. The Linear Road benchmark stresses the performance of a continual query system, in our case STREAM [1], as it executes queries that calculate real-time tolls. The application is sensitive to latencies since it must receive data and return answers to drivers. To calculate the tolls, the query engine must receive and evaluate data points from simulated vehicles on the highway to calculate traffic flow volume, from which the toll is set. In our version, we added adaptive QoS mechanisms to react to out-of-bounds latency conditions that reduced latencies and allowed for greater system utilizations [25].

As a third test of our code generation architecture, we have used ACCT in benchmarks with Cauldron and SmartFrog evaluating the complete toolkit. In our tests, we compared using the generated SmartFrog deployment of a 3-tier application to deployment using only hand-written scripts. While this tool is still in the early phase of its development, the generated plan matched the hand-written deployment plan for startup performance time, but significantly, provably met deployment constraints whereas no such statement could be made for the handcrafted script.

Table 3. Lines of C code contributed by each AXpect template in the streaming video application [26]

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

6. RELATED WORK

Most closely related to the architecture of our code generator is that it adopts a similar architecture already used by compilers. Also, it adopts an intermediate format for flexibility like `gcc` and Flick [11]. However, there are several important features. Traditional compilers only map into basic assembly code. Flick, too, is restricted in its ability to output because it does not maintain a system state document as we do with XIP. This is crucial in achieving the flexibility to do code weaving. SourceWeave.NET is also similar in that it is a cross-platform weaver.

The Polyglot project has focused on creating extensible high-level languages [18]. However, while Polyglot has seen use in other projects, users are limited to variants on Java syntax whereas our architecture permits the use of any human-friendly syntax which can then be compiled to an XML intermediate format.

The SoftArch/MTE [13] and Argo/MTE [7] projects have also used XML + XSLT for code generation. Their project has primarily concerned with resolving mismatches between software engineering tools. Our results corroborate their experience. In addition, we go significantly beyond this and use Clearwater in the ISG as a DSL implementation technique and for aspect weaving.

7. CONCLUSION

Based on our experience, using XML technologies in code generation efforts can be extremely beneficial. We have described our general architecture and given two examples of generators employed by our research group that illustrate the ability of this technique to accommodate a variety of implementation languages and a variety of input languages.

When we generate a new document by using XML, we are able to express the semantic structure as inherited from in the higher layers of abstraction – the Spi or XIP document. In computer science theory, it is well-known that it is impossible to prove the equivalence of two programs – it is impossible for any computer program to "understand" another program. However, maintaining this domain information means that instead of understanding the general purpose language that has been generated, our code generator, and any later stages, need only operate on source code performing specific tasks taken from our domains.

Our future research plans are to expand the weaver capabilities to the system level from just source-level weaving for the ISG. This work would also include exploration for new ways to write the encode aspects for the AXpect module so that they are more readable. We anticipate this being a valuable architecture for implementing multiple domain specific languages that encode differing aspects of information flow systems. Also, we are proceeding on with goals 2 and 3 of ACCT, and there may well be some integration work done between the two efforts in the future.

Finally, it is worth noting that while we have encountered much of the important technology in XSLT we are also investigating the Apache Software Foundation's DVSL [10], a scripting language based on Velocity, for code generation, also, as it promises enhanced readability over XSLT.

8. ACKNOWLEDGEMENTS

This work was partially supported by NSF/CISE IIS and CNS divisions through grants IDM-0242397 and ITR-0219902, DARPA ITO and IXO through contract F33615-00-C-3049 and N66001-00-2-8901, and Hewlett-Packard. Also, we wish to thank the anonymous reviewers for their helpful comments and suggestions.

9. REFERENCES

[1] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J.,

STREAM: The Stanford data stream management system. In *Data Stream Management: Processing High-Speed Data Streams*. Garofalakis, M., Gehrke, J., and Rastogi, R., eds. To appear 2006. Springer

[2] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R., Linear Road: A Stream Data Management Benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, August, 2004.

[3] Barreto, L., Douence, R., Muller, G., and Südholt, M., 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.

[4] Birrell, A., and Nelson, B. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems, 2, 1* (Feb. 1984), 39-59. Also appeared in Proceedings of SOSP'83.

[5] Black, P., Huang, J., Koster, R., Walpole, J., and Pu, C. Infopipes: an abstraction for multimedia streaming. *ACM Multimedia Systems Journal*, 8(5): 406-419, 2002.

[6] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cohan, J., eds. *Extensible Markup Language*. http://www.w3.org/TR/xml11. World Wide Web Consortium (W3C). 2004.

[7] Cai, Y., Grundy, J., and Hosking, J. Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. ASE 2004.

[8] Clark, J. ed. *XSL Transformations*. http://www.w3.org/TR/xslt. World Wide Web Consortium (W3C). 1999.

[9] Clark, J., and De Rose, S., eds. *XML Path Language* http://www.w3.org/TR/xpath. World Wide Web Consortium (W3C). 1999.

[10] DVSL. http://jakarta.apache.org/velocity/dvsl/

[11] Eide, E., Frei, K., Ford, B., Lepreau, J., and Lindstrom, G. Flick: a flexible, optimizing IDL compiler. In *Proceedings of the 1997 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)* (Las Vegas, NV, Jun 15-18, 1997).

[12] Eisenhauer, G., Bustamente, F., and Schwan, K. A middleware toolkit for client-initiated service specialization. *Proceedings of the PODC Middleware Symposium* – (Portland Oregon, July 18-20, 2000).

[13] Grundy, J., Cai, Y., and Liu, A. SoftArch/MTE: generating distributed system test-beds from high-level software architecture descriptions. In the *Proceedings of ASE 2001: The 16th IEEE Conference on Automated Software Engineering*. (Coronado, CA, November 26-29, 2001).

[14] Karsai, G. Why XML is not suitable for semantic translation. Research note, Nashville, TN, April, 2000.

[15] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In the *Proceedings of European Conference on Object-Oriented Programming, Aspect-Oriented Programming Workshop (ECOOP '97).* (Jyväskylä, Finland, June 10, 1997.)

[16] Le Hégaret, P., DOM Activity Lead. *Document Object Model (DOM)*. http://www.w3.org/DOM/. World Wide Web Consortium (W3C).

[17] Lieberherr, K. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[18] Nystrom, N., Clarkson, M. R., and Myers, A. C. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction* (Warsaw, Poland, April 2003). Springer-Verlag LNCS 2622, 138–152.

[19] Pu, C., Schwan, K., and Walpole, J. Infosphere project: system support for information flow applications. *ACM SIGMOD Record, 30, 1* (Mar. 2001), 25-34.

[20] Pu, C., and Swint, G. DSL weaving for distributed information flow systems (Invited Keynote). *Proceedings of the 2005 Asia Pacific Web Conference. (APWeb05)*. (Shanghai, China. March 29 - April 1, 2005.) Springer-Verlag LNCS. 2005.

[21] Sahai, A., Pu, C., Jung, G., Wu, Q., Yan, W., and Swint, G. Towards automated deployment of built-to-Order systems. In *Proceedings of the 16th IFIP/IEEE Distributed Systems: Operations and Management (DSOM '05*) (Barcelona, Spain. October 24-26, 2005). To appear.

[22] Sahai, A., Joshi, R., Singhal, S., and Machiraju, V. Automated policy based resource construction in utility computing environments. In the *Proceedings of the 2004 IEEE/IFIP Network Operations & Management Symposium (NOMS 2004)*. (Seoul, Korea. April 19-24, 2004.)

[23] Salle, M., Sahai, A., Bartolini, C., and Singhal, S. A business-driven approach to closed-loop management. HP Labs Technical Report HPL-2004-205, November 2004.

[24] Sarkar, S. Model driven programming using XSLT: an approach to rapid development of domain-specific program generators. www.XML-JOURNAL.com. August 2002.

[25] Swint, G., Jung, G., and Pu, C. Event-based QoS for a distributed continual query system. *The 2005 IEEE International Conference on Information Reuse and Integration (IRI 2005)* (Las Vegas, Nevada. August 14-17, 2005).

[26] Swint, G., and Pu, C. Code generation for WSLAs using AXpect. *Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004)* (San Diego, California. July 6-9, 2004).

[27] Swint, G., Pu, C., Koh, Y., Liu, L., Yan, W., Consel, C., Moriyama, K., and Walpole, J. Infopipes: The ISL/ISG Implementation Evaluation. *Proceedings of the 3rd IEEE Network Computing and Application Symposium 2004 (IEEE NCA04)*. (Cambridge, Massachusetts. August 30 - September 2, 2004.)

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