XML Pipelining for Mathematical Computation

R. Alexander Milowski
Mathematics Department
University of California, Davis
milowski@math.ucdavis.edu

1. Overview

Exposing a single computation on the Internet via a web service can be challenging. To do so, the input and output of such a service needs to be codified, typically in an XML Schema, and code needs to be written to translate to and from XML. Further, computations need to be interfaced and run with their results formulated into XML. To complicate matters further, there is the whole process of packaging such a system for deployment of some web services platform.

While many web services frameworks focus on the interfaces, what is at the core of exposing a web service is the processing of an XML vocabulary (the request elements). Each vocabulary is associated with a set of actions–implicitly or explicitly–to make a wanted computation to take place. Unfortunately, XML isn't always that simple to process and usually this involves writing a "pile of code". This forces the development of web services to be only in the domain of the expert programmer and not that of a Research Mathematician.

Further, what then happens when this service needs to deployed into other environments that aren't necessary a "web service"? The computation has the same characteristics of input and output, but the packaging and connections are different. Such services should be packaged just as XML services and their "web services" versions should just be bindings of that to web protocols.

In many ways, these problems are very similar to the description of the "Programming Challenge" for computational sciences as described by Douglas Post [16]. He describes a very similar challenge in writing massive parallel algorithms for grid computing where the desire to build a system is overwhelmed by the programming necessary to implement the system. Certainly, XML-based web services can be part of such grid systems yet very little attention has been paid to the challenges of processing XML or packaging XML services.

For XML, there are multitudes of XML-related standards for representing information and for processing XML to accomplish a single task. Many parts of implementing a web service involve using these technologies for specific purposes. Further, any web service request can be reduced to a sequence of "application steps" where some of the steps are XML technologies like XSLT[5] for transformation, XML Schema [19] for validation, XQuery [2] for queries, etc. But there are no standards for how these technologies are orchestrated into an XML process.

This paper proposes utilizing a concept called XML Pipelines for implementing XML-based mathematical computations. In short, an XML pipeline provides a way to chain together the processing steps and, by doing so, simplifies the integration of XML technologies with non-XML components. Ultimately, utilizing an XML pipeline separates concerns and encapsulates complexity–which reduces necessary skill level of the user. They also facilitate deploying XML processes as web services.

This paper is distinctively not about SOAP[7], WSDL[4], and the myriad of other web service "standards." Those so-called standards are about bindings to protocols on the outside of a service. The technique described here is about the processing that happens inside an XML service. That is, when you start building your XML service, the focus of this paper is the steps and techniques that let you actual implement the processing and connect disparate technologies.

2. A Simple Example

The following is a simple example of exposing a computation:

(1) A developer would like to expose the ability to compute the reduced Gröbner basis of a toric ideal of a matrix for a given monomial order. This is really a two step process: first we compute an integer basis of the kernel of the matrix and, second, we compute the Gröbner basis using this kernel basis as input via the algorithm described in [17].
(2) This computation will be exposed as a web service where a matrix is sent in XML and the basis comes back in XML.
(3) While the request format is constant, the result may not be. It would be really useful to get the basis formatted in a number of ways: MathML[3], the original request markup, and possibly other XML formats.

A natural place to start for the request document is MathML as the matrix can easily be encoded in MathML:

```
<matrix xmlns="http://www.w3.org/1998/Math/MathML">
<matrixrow><cn>1</cn><cn>2</cn><cn>3</cn>...</matrixrow>
...
</matrix>
```

While verbose, it captures almost every aspect of this matrix. Unfortunately, it can't encode the remaining request parameters (e.g monomial order, output format,...). What is missing is the context of the request.

It isn't sufficient just to encapsulate the data as the data has context. That is, the context must be present as well:

- Is it a matrix or does it represent a binomial ideal?
- Is the matrix rational or integer?
- Does it represent a Gröbner basis or does one need to be computed?
- What are the specifics of the computation (e.g. monomial ordering, output format, etc.).

Since this is XML, the context and the data can easily be mixed. To simplify the data, we'll use a custom matrix element that encodes the matrix rows as a list of integers (something that XML Schema can easily type). The rest of the information we can encode as children or attributes of the main request element:

```
<m:compute-toric-ideal xmlns:m="http://www.milowski.com/schemata/monos/2005"
   monomial-order="wgrevlex">
<m:matrix>
<m:ilist>1 2 3 4 0 1 4 5</m:ilist>
<m:ilist>2 3 4 1 1 4 5 0</m:ilist>
<m:ilist>3 4 1 2 4 5 0 1</m:ilist>
<m:ilist>4 1 2 3 5 0 1 4</m:ilist>
</m:matrix>
<m:weight>1 1 1 1 1 1 1 1</m:weight>
<m:row-space-vector>10 10 10 10 10 10 10 10</m:row-space-vector>
</m:compute-toric-ideal>
```

Here we have wrapped the entire message with an element that specifies monomial order, contains the integer matrix, and contains the weight vector to be used for the graded monomial ordering.

It should be noted that MathML could have been used for part of the request but the custom markup does a better job of encapsulating the whole thing. In this case, the matrix is an integer matrix and there is no efficient or direct representation of such an object in MathML. Further, mixing markup vocabularies in such a small request makes the process of invoking the service much more cumbersome.

For the return, different outputs might need to be supported. Here again we can use context to our advantage. Consider that as the request is specified, we have everything necessary to output another matrix that represents a set of binomials. Further, we could create a 'binomial' element and output a set of binomial elements as well. Even further, we might want MathML output for presentational uses.

The simplest solution is to make the service act as an XML filter according to the following rules:

(1) Any [m:]compute-toric-ideal element is processed into a Gröbner basis for the toric ideal.
(2) If the [m:]compute-toric-ideal element is contained by a [m:]output-format element, then the result will be processed into the requested format.

Given the above, the following request would produce the Gröbner basis for the toric ideal formatted as MathML markup:

```
<m:output-format type="mathml">
<m:compute-toric-ideal xmlns:m="http://www.milowski.com/schemata/monos/2005"
   monomial-order="wdegrevlex" output="mathml">
```

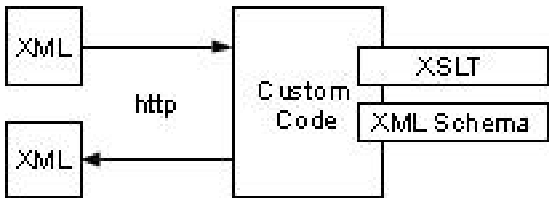

FIGURE 1. Simple Service Architecture

```
...
</m:compute-toric-ideal>
</m:output-format>
```

All of this sounds reasonable but the reality is that software isn't usually written to be this flexible. Even further, we haven't gotten to how the computation is actually performed. But it is very important that services are flexible as their use will extend beyond the original intent.

Let's start with some simple assumptions and assume that there are two different software components that can consume the [m:]matrix elements and output an [m:]matrix. One of these software components computes the kernel of the integer matrix and the other computes a basis of the toric ideal from a kernel basis.

The necessary steps for this web service are:

(1) Inspect the input for the monomial ordering and to retrieve the matrix.
(2) Compute the kernel basis from the matrix XML and produce a new XML matrix element representing that basis.
(3) Compute the basis from the result of (2) to produce a matrix element representing that Gröbner basis.
(4) If the output requested is 'mathml', transform the matrix XML into appropriate MathML.

Fortunately, XSLT can be used to accomplish most of (4) but the whole process needs to be orchestrated by writing custom code that interfaces the different APIs of the software components. Further, the different XML inputs and outputs need to be handled properly to ensure the output of the whole process is both correct and well-formed XML. Even further, we might want to validate the XML using XML Schema to make sure the inputs and outputs are correct.

That last bit is a critical point in that once you start using XML to encode information you'll want to use companion technologies to manipulate that XML. A whole suite of important technologies and standards have been developed to address issues of validity, transformation, query, etc. It is important that a service be able to intermix these technologies with their own code.

Without any additional support, the system architecture for this web service is show in figure 1. Here we have a request document that is received over some Internet protocol (i.e. HTTP) which is then passed to a "pile of code". This code may use XSLT, XML Schema, or any other XML-related processing technologies. The code is responsible for computing the requested computation, orchestrating the XML processing and production, and for returning the result serialized into XML syntax.

Even for a trivial computation, orchestrating the mix of XML technologies, standards, APIs, and custom code is a daunting task. The interconnect between such technologies is non-trivial and may require the developer to become an expert in coding to specific API. As a result, a developer may choose not to use different technologies–no matter how appropriate–due to the time involved to integrate them. In essence, flexible has been destroyed by the diversity of technologies–which is counter to one of the reasons for originally using XML.

A stepwise process was described In the original description the service and good programmer might structure their code the same way. This would make the service less fragile and so why not structure the whole process stepwise? The service can be viewed as a pipeline of steps where each output is chained to the input of the next step–much like function composition. The difference is that each step is required to produce and consume XML.

Given this idea, the service can be restructured as in figure 2. Here the specification of the processing matches the service architecture. Fortunately, there is a forthcoming technology brewing in the XML industry called XML Pipelining

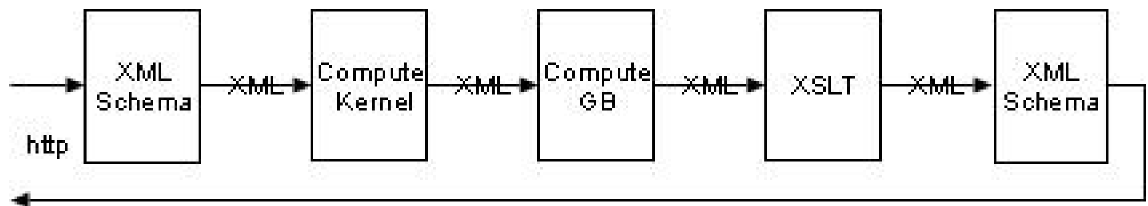

Figure 2. Simple XML Pipeline

that lets this process be described without coding. This helps solve the interconnect problem and lessens the "Programming Challenge".

3. What is an XML Pipeline?

The concept of an XML Pipeline originates in Computational Linguistics [10] where it was used for Named Entity recognition. In short, each component of the pipeline was part of a "fat pipe" where the input and outputs were XML. For linguistics, each step did one task well: parsing paragraphs in sentences, identifying parts of speech, disambiguating parts of speech, categorizing terms. The ability to recombine each tool very quickly allowed the linguistic research to build new pipelines very fast without programming. Also, as each step added more information to the result so that later steps could do their job better.

Since that time, XML pipelining has shown up in many different technologies: smallx XML Pipelines [14], Sun XML Pipeline Note[21], SXPipe[20], Markup Technology's Pipelining Server[18], and Apache Cocoon[1]. While that list is non-exhaustive, it does show how XML processing can be easily viewed as a step-wise process. The difference between these technologies and other XML processing frameworks is that the XML pipeline is a first-class object represented by an XML vocabulary (document). Thus, it can be authored without knowledge of programming APIs.

In example 1, the service described in the previous section was encoded as smallx XML pipeline. This pipeline vocabulary allows each step to have its own vocabulary element for describing the steps–where many of the XML technologies are built-in to the pipelining engine. In addition, custom steps can be easily integrated with their own step vocabulary elements.

The first step of this pipeline is a validation step where the input document is validated against a schema. This ensures the assumptions of the pipeline are correct as the input is valid. The attribute 'assert' with a value of 'true' tells the pipeline to fail if the input is not valid.

Next, the 'monos' prefixed elements represent steps processed by the Monos algebra software[12]. This first step named [monos:]kernel-basis will calculate an integer kernel of the input matrix. The second step, [monos:]binomial-groebner will calculate the Gröbner basis of the toric ideal. The output of that step is another matrix.

The penultimate step of this pipeline is to transform the desired result into MathML if that was requested. Here the input still has the wrapper m:output-format element that will either be removed via the XSLT transform or via the [p:]unwrap step. At the end, the last step is validation to ensure the output is correct from an XML perspective.

Example 1. ***Toric Ideal Example***

```
<p:pipe name="compute-gb"
   xmlns:p="http://www.smallx.com/schemata/pipeline/2005"
   xmlns:monos="http://www.milowski.com/schemata/monos/steps/2005"
   xmlns:m="http://www.milowski.com/schemata/monos/2005"
   extension-prefixes="monos"
>
  <!-- Validate the input -->
  <p:validate assert="true">
    <p:map namespace="http://www.milowski.com/schemata/monos/2005"
           href="monos-vocabulary.xsd"/>
  </p:validate>
```

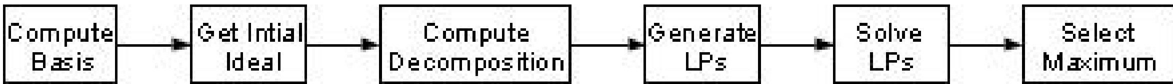

FIGURE 3. Computation Steps

```
    <!-- Compute the kernel of the matrix -->
    <monos:kernel-basis/>

    <!-- Compute the basis from the kernel -->
    <monos:binomial-groebner/>

    <!-- Format the results -->
    <p:route>
      <p:when test="/m:output-format[@type='mathml']">
        <p:xslt src="ideal2mathml.xsl"/>
      </p:when>
      <p:when test="/m:output-format[@type!='mathml']">
        <p:unwrap select="m:output-request"/>
      </p:when>
    </p:route>

    <!-- Validate the results -->
    <p:validate assert="true">
      <p:map namespace="http://www.milowski.com/schemata/monos/2005"
             href="monos-vocabulary.xsd"/>
      <p:map namespace="http://www.w3.org/1998/Math/MathML"
             href="mathml.xsd"/>
    </p:validate>
  </p:pipe>
```

It should be noted that nothing about XML pipelines require that the XML actually be serialized. Smart pipelining implementations can pass whole XML infosets [6] between steps when the components are tightly integrated. In fact, XML infosets can be streamed through the pipeline to allow processing of large documents.

Further, no aspects of the binding protocols has entered our processing. There is no SOAP message envelope to discard, WSDL to interact with, nor any other complication. The process is specified purely in terms of the process of the "request body" (i.e. the request itself).

This encapsulation of the processing is an important feature of XML Pipelining. The pipeline can be re-purposed into new environments without changes. This allows deployment into protocol-neutral environments like the JXTA P2P platform[9].

4. Computing Integer Programming Gaps with an XML Pipeline

A more complicated example of computing with an XML pipeline is the process of computing an Integer Programming Gap as described in [11]. The process takes as input a matrix coming from an Integer Program along with a cost vector. The result of the computation is the gap (a number) between the linear approximation and the integer solution. This computation is a multi-step process where the data from one step feeds the next and so is a suitable example of using pipelines to compute Mathematics.

The stepwise view of this gap computation is shown in figure 3. While this captures the stepwise process, it does not capture the view of the data produced–which is much more important. This data flow is shown in figure 4.

The full version of this pipeline is shown in example 2. This pipeline runs through each computation step as detailed above and the result is the annotated component of the monomial ideal decomposition that produced the maximum gap. It should be noted that there was hidden complexity in this computational pipeline as there are 11 steps in the pipeline and

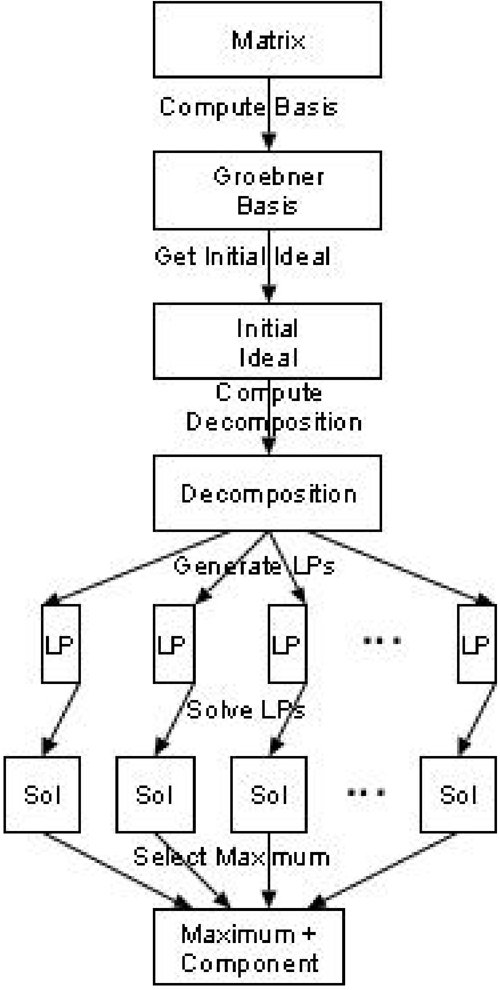

FIGURE 4. Computation Data Flow

6 steps in figure3. Some of this complexity comes from the technology components, some from the request's flexibility, and some from branching to handle error states.

Example 2. ***Integer Gap Pipeline***

```
<p:pipe name="integer-gap"
   xmlns:p="http://www.smallx.com/schemata/pipeline/2005"
   xmlns:m="http://www.milowski.com/schemata/monos/2005"
   xmlns:monos="http://www.milowski.com/schemata/monos/steps/2005"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   extension-prefixes="monos"
>

<!-- Step 1 -->
<p:let>
   <p:parameter name="matrix" select="m:matrix"/>
   <p:parameter name="weight" select="m:weight"/>
   <p:sequence>
      <!-- Step 2 -->
      <p:xslt>
         <xsl:transform version="1.0">
            <xsl:template match="m:gap-matrix">
               <m:compute-toric-ideal monomial-order="wgrevlex" output="matrix-compact">
                  <xsl:copy-of select="*"/>
                  <xsl:if test="not(m:row-space-vector)">
                     <m:compute-row-space-vector>
                     <xsl:copy-of select="m:matrix"/>
                     </m:compute-row-space-vector>
                  </xsl:if>
               </m:compute-toric-ideal>
            </xsl:template>
         </xsl:transform>
      </p:xslt>
      <!-- Step 2.1 -->
      <p:subtree select="m:compute-toric-ideal/m:compute-row-space-vector">
         <monos:column-sum-matrix/>
         <p:unwrap select="m:ilist"/>
         <p:rename select="m:compute-row-space-vector" name="m:row-space-vector"/>
      </p:subtree>

      <!-- Step 3 -->
      <monos:kernel-basis/>

      <!-- Step 4 -->
      <p:route>

        <!-- Step 4.1 -->
        <p:when test="m:matrix/m:ilist">

          <!-- Step 5 -->
          <monos:binomial-groebner/>

          <!-- Step 6 -->
          <monos:initial-ideal/>

          <!-- Step 7 -->
          <p:wrap name="m:decompose"/>
          <monos:decompose/>

          <!-- Step 8 -->
          <p:xslt>
             <xsl:transform version="1.0">
                <xsl:param name="matrix"/>
                <xsl:param name="weight"/>
                <xsl:template match="/">
                   <m:formulate-gap>
```

```
                    <xsl:copy-of select="$matrix"/>
                    <xsl:copy-of select="$weight"/>
                    <xsl:copy-of select="."/>
                 </m:formulate-gap>
              </xsl:template>
           </xsl:transform>
        </p:xslt>

        <!-- Step 9 -->
        <monos:formulate-gap-lp/>

        <!-- Step 10 -->
        <monos:solve-lp/>

        <!-- Step 11 -->
        <p:route>

          <!-- Step 11.1 -->
          <p:when test="m:error">
             <p:document>
             <m:error>Could not solve all the linear programs which is probably due to bad data or program error.
             </p:document>
          </p:when>

          <!-- Step 11.2 -->
          <p:otherwise>

            <p:xquery>
            declare namespace m = "http://www.milowski.com/schemata/monos/2005";
            declare namespace gap = "http://www.milowski.com/example";

            declare function gap:convert-rational($v) as xs:double {
               if (contains($v,"/"))
                  then number(substring-before($v,"/")) div
                       number(substring-after($v,"/"))
                  else number($v)
            };

            (for $sol in /m:component-lp-list/m:component-lp/m:solution
                          order by gap:convert-rational($sol/@value) descending
               return $sol)[1]/..
            </p:xquery>
          </p:otherwise>
        </p:route>
    </p:when>

    <!-- Step 4.2 -->
    <p:otherwise>
      <p:document>
        <m:error>Kernel is empty.</m:error>
      </p:document>
    </p:otherwise>
  </p:route>

</p:sequence>

</p:let>

</p:pipe>
```

The following is a description of how the pipeline executes. Each step is describe and an example of the output is shown (except for step 1). Each example assumes the 'm' prefix is bound to "http://www.milowski.com/schemata/monos/2005".

(1) The simplest input to the pipeline is a document as follows:

```
<m:gap-matrix>
<m:matrix>
<m:ilist>0 0 1 1 2 3</m:ilist>
<m:ilist>0 1 0 1 0 0</m:ilist>
<m:ilist>1 0 0 0 0 0</m:ilist>
</m:matrix>
<m:weight>1 1 1 1 1 1</m:weight>
</m:gap-matrix>
```

The very first thing that is done is the parameters 'matrix' and 'weight' are bound to the [m:]matrix and [m:]weight elements, respectively. This binding will be in effect for the body of the [p:]let step. There is an optional [m:]row-space-vector that can be specified for the computation.

(2) The input is converted into a toric ideal basis computation request by a combination of renaming the element and setting an attribute. At step 2.1, if the row-space vector was not specified, it is computed from a second copy of the matrix. The result is:

```
<m:compute-toric-basis monomial-order="wgrevlex">
<m:matrix>
<m:ilist>0 0 1 1 2 3</m:ilist>
<m:ilist>0 1 0 1 0 0</m:ilist>
<m:ilist>1 0 0 0 0 0</m:ilist>
</m:matrix>
<m:weight>1 1 1 1 1 1</m:weight>
<m:row-space-vector>1 1 1 2 2 3</m:row-space-vector>
</m:compute-toric-basis>
```

Here it should be noted that we have mixed pure XML manipulation with a component that can compute column sums of a matrix.

(3) The kernel of the basis is computed by the [monos:]kernel-basis step. This step only operates on the contained [m:]matrix element and will replace that element with a matrix representing the kernel.

```
<m:compute-toric-basis monomial-order='wgrevlex'>
<m:matrix>
<m:ilist>0 -1 -1 1 0 0</m:ilist>
<m:ilist>0 0 -1 0 -1 1</m:ilist>
<m:ilist>0 1 -1 -1 1 0</m:ilist>
</m:matrix>
<m:weight>1 1 1 1 1 1</m:weight>
<m:row-space-vector>1 1 1 2 2 3</m:row-space-vector>
</m:compute-toric-basis>
```

(4) This step doesn't change anything but routes the content depending on whether the kernel is empty. This routing depends on the XML content of the kernel represented as a matrix. In theory, no request matrix should have an empty kernel but a user could easily make that mistake. This situation is detected and the computation ends with an error document.

(5) The Gröbner basis of the toric ideal is computed by the [monos:]binomial-groebner step and is output as a matrix.

```
<m:binomial-ideal>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:matrix>
<m:ilist>0 0 -1 0 2 -1</m:ilist>
<m:ilist>0 -1 0 1 1 -1</m:ilist>
<m:ilist>0 0 1 0 1 -1</m:ilist>
<m:ilist>0 -1 1 1 -1 0</m:ilist>
<m:ilist>0 0 2 0 -1 0</m:ilist>
<m:ilist>0 1 1 -1 0 0</m:ilist>
<m:ilist>0 2 0 -2 1 0</m:ilist>
<m:ilist>0 3 0 -3 0 1</m:ilist>
```

```
</m:matrix>
</m:binomial-ideal>
```

(6) The initial ideal of the Gröbner basis is computed by the [monos:]initial-ideal step.

```
<m:monomial-ideal>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:monomial>0 0 0 0 2 0</m:monomial>
<m:monomial>0 0 0 1 1 0</m:monomial>
<m:monomial>0 0 1 0 1 0</m:monomial>
<m:monomial>0 0 1 1 0 0</m:monomial>
<m:monomial>0 0 2 0 0 0</m:monomial>
<m:monomial>0 1 1 0 0 0</m:monomial>
<m:monomial>0 2 0 0 1 0</m:monomial>
<m:monomial>0 3 0 0 0 1</m:monomial>
</m:monomial-ideal>
```

(7) The monomial ideal is first wrapped with a [m:]decompose element so that the decomposition is computed by the [monos:]decompose step.

```
<m:lexed-list>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:monomial-ideal>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
<m:monomial>0 0 0 0 0 1</m:monomial>
</m:monomial-ideal>
<m:monomial-ideal>
<m:monomial>0 3 0 0 0 0</m:monomial>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
</m:monomial-ideal>
<m:monomial-ideal>
<m:monomial>0 2 0 0 0 0</m:monomial>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 1 0 0</m:monomial>
<m:monomial>0 0 0 0 2 0</m:monomial>
</m:monomial-ideal>
<m:monomial-ideal>
<m:monomial>0 1 0 0 0 0</m:monomial>
<m:monomial>0 0 2 0 0 0</m:monomial>
<m:monomial>0 0 0 1 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
</m:monomial-ideal>
</m:lexed-list>
```

(8) The input to the step that formulates the linear programs for the gap computation is created via XSLT. This transformation uses the parameters 'matrix' and 'weight' to get content from the original request. The output is an aggregate of that information.

```
<m:formulate-gap>
<m:matrix>
<m:ilist>0 0 1 1 2 3</m:ilist>
<m:ilist>0 1 0 1 0 0</m:ilist>
<m:ilist>1 0 0 0 0 0</m:ilist>
</m:matrix>
<m:weight>1 1 1 1 1</m:weight>
<m:lexed-list>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:monomial-ideal>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
<m:monomial>0 0 0 0 0 1</m:monomial>
```

```
</m:monomial-ideal>

... more components ...

</m:lexed-list>
</m:formulate-gap>
```

This step is another example of using standard XML technologies to transform data for input to mathematical computations.

(9) The LP programs are formulated via the [monos:]formulate-gap-lp step. For each component of the decomposition, a linear program is generated and the component (monomial ideal) is listed for traceability.

It should be noted that in the original data flow diagram for this pipeline a set of linear programs was generated. This translates to a set of sibling [m:]component-lp elements. As the pipelines can stream the XML infoset, these components can be processed individually by the following steps.

```
<m:component-lp-list>
<m:component-lp constant='0'>
<m:monomial-ideal>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
<m:monomial>0 0 0 0 0 1</m:monomial>
</m:monomial-ideal>
<m:linear-program>
H-representation
begin
   9 7 rational
   0 0 0 1 0 0 0
   0 0 0 0 0 1 0
   0 0 0 0 0 0 1
   0 0 0 -1 -1 -2 -3
   0 0 0 1 1 2 3
   0 0 -1 0 -1 0 0
   0 0 1 0 1 0 0
   0 -1 0 0 0 0 0
   0 1 0 0 0 0 0
end
maximize
   0 -1 -1 -1 -1 -1 -1
</m:linear-program>
<m:c>1 1 1 1 1 1</m:c>
<m:u>0 0 0 0 0 0</m:u>
<m:b>0 0 0</m:b>
</m:component-lp>

... more component lps ..

</m:component-lp-list>
```

(10) This step is an example of a filter step in that each linear program is solved by processing each [m:]linear-program element while leaving the rest of the document unchanged. The step replaces the [m:]linear-program elements with an [m:]solution element representing the solution to the linear program.

```
<m:component-lp-list>
<m:component-lp constant='0'>
<m:monomial-ideal>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
<m:monomial>0 0 0 0 0 1</m:monomial>
</m:monomial-ideal>
```

```
<m:solution value='0'>
<m:point>0 0 0 0 0 0</m:point>
</m:solution>
<m:c>1 1 1 1 1 1</m:c>
<m:u>0 0 0 0 0 0</m:u>
<m:b>0 0 0</m:b>
</m:component-lp>

... more component lps ...
</m:component-lp-list>
```

(11) Finally, we check for errors. If there are no errors, amongst all the solutions, the [m:]component-lp element with the largest 'value' attribute on the contained [m:]solution element is selected. This element represents the largest integer gap and is the result of the pipeline. In this step, another XML technology called XQuery [2] is used to query the resulting documents and select the correct element.

```
<m:component-lp constant='2'>
<m:monomial-ideal>
<m:lex-order>x1 x2 x3 x4 x5 x6</m:lex-order>
<m:monomial>0 3 0 0 0 0</m:monomial>
<m:monomial>0 0 1 0 0 0</m:monomial>
<m:monomial>0 0 0 0 1 0</m:monomial>
</m:monomial-ideal>
<m:solution value='2/3'>
<m:point>0 0 0 2 0 -2/3</m:point>
</m:solution>
<m:c>1 1 1 1 1 1</m:c>
<m:u>0 2 0 0 0 0</m:u>
<m:b>0 2 0</m:b>
</m:component-lp>
```

While the gap computation produces a single number as a result, it is important that the component of the decomposition of the monomial ideal also be identified. In fact, in the above example, the result of the pipeline is the gap value, the solution, the vectors and constants used to formulate the linear program, and the monomial ideal decomposition component. These critical bits of information are all output by the pipeline so that they might be used to further analyze the result.

Ideally, it would be nice to have the whole computation be traceable in reverse from an interesting result. This is easily accomplished as each step produces an XML document and so could be written to a resource. In fact, wrapping computations with metadata using some vocabulary like ATOM [15] would allow encapsulation of the software computed the result, when it was computed, by whom, along with the data the steps produced. In the end, the whole computation is an ATOM news feed and can be read by a news reader. While this might make the pipeline more complicated, it will save time for the user in terms of traceability, analysis, and debugging of results.

The essential fact to take from this is that programming such orchestrations and manipulations is very onerous. XML Pipelines solve both the interconnect and XML processing problems. They also lend themselves to extension as it is very easy to add "one more step" and adjust the output in some critical way (e.g producing MathML or validating input/output). Pipelines lend themselves naturally to extension because the interconnect between components is XML and so very simple to handle.

Given that the XML technologies can easily be mixed with non-XML components, vocabularies like ATOM [15] and metadata standards like the Dublin Core [8] can be used to store results. This allows program independent views of mathematical objects to be stored along with their essential metadata about those results (i.e. data + context). All this naturally flows from the use of XML but is easy to implement within an XML pipeline.

5. Current and Future Work

There's a crucial reason for putting mathematical computations into pipelines rather than just web services is re-purposeable computing. Each XML pipeline represents an XML process that can accomplish a mathematical computation without reference to where it is to be deployed. The XML pipeline can be deployed into many different contexts such as a web service or via a menu item in your favorite mathematics software suite.

Examples of XML pipelines as web services can be see online via the Monos demo [13]. This demonstration is a web application (a set of forms) that interfaces a set of web services–both of which were built using XML pipelines. Each web service runs a pipeline that accepts XML to compute Mathematical objects via Monos. The results can be viewed in several different formats.

Ultimately, the purpose of integrating the smallx XML pipelines and Monos algebra software was beyond just exposing web services. The author has developed a new computational grid software that utilizes an JXTA-based[9] peer-to-peer (P2P) infrastructure to distributed computations. Each peer communicates to each other with XML over the P2P network and performs their computations on messages by running XML pipelines.

In summary, when a computation is described as an XML pipeline, it can be moved from a request-response environment like a web service, to a P2P environment , and then to the desktop without change. The processing of the XML is encapsulated by the pipeline and all that needs to change is the binding of that pipeline into different environments. This separation of concerns allows re-purposing of processing independent of deployment environment.

References

[1] Apache. Cocoon. http://cocoon.apache.org/, 2004.
[2] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/, 2005.
[3] D. Carlisle, P. Ion, R. Miner, and N. Poppelier. Mathematical markup language (mathml) version 2.0 (second edition). http://www.w3.org/TR/MathML2/, 2004.
[4] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. http://www.w3.org/TR/wsdl20/, 2005.
[5] J. Clark. Xslt 1.0. http://www.w3.org/TR/xslt, 1999.
[6] J. Cowan and R. Tobin. Xml information set (second edition). http://www.w3.org/TR/xml-infoset/, 2004.
[7] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, 2003.
[8] D. C. M. Initiative. Atom. http://dublincore.org/documents/dc-xml-guidelines/, 2003.
[9] B. Joy and M. Clary. Jxta. http://www.jxta.org, 2005.
[10] A. Mikheev, C. Grover, and M. Moens. Xml tools and architecture for named entity recognition. *Journal of Markup Languages: Theory and Practice*, 1(3):89–113, 1999.
[11] R. A. Milowski. Computing irredundant irreducible decompositions and the scarf complex of large scale monomial ideals. Master's thesis, San Francisco State University, 2004.
[12] R. A. Milowski. Monos algebra software. http://www.milowski.com/software/monos, 2005.
[13] R. A. Milowski. Monos web demo. http://www.milowski.com/app/monos-demo/, 2005.
[14] R. A. Milowski. smallx. http://smallx.dev.java.net, 2005.
[15] M. Nottingham and R. Sayre. Atom. http://www.ietf.org/html.charters/atompub-charter.html, 2005.
[16] D. Post. Expert opinion: The coming crisis in computational science. *High Performance Computing*, 13(11), March 2004.
[17] B. Sturmfels. *Groebner Bases and Convex Polytopes*. American Mathematical Society, 1995.
[18] H. S. Thompson. Mt pipeline. http://www.markup.co.uk/, 2005.
[19] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. Xml schema part 1: Structures second edition. http://www.w3.org/TR/xmlschema-1, 2001.
[20] N. Walsh. Simple xml pipelines (sxpipe). https://sxpipe.dev.java.net/, 2004.
[21] N. Walsh and E. Maler. Xml pipeline definition language version 1.0. http://www.w3.org/TR/2002/NOTE-xml-pipeline-20020228/, 2002.