

# An Orchestration for Distributed Web Service Handlers

Beytullah Yildiz<sup>1,2</sup>, Shrideep Pallickara<sup>1</sup>, Geoffrey Fox<sup>1,2,3</sup>

<sup>1</sup>*Community Grids Lab, Indiana University*

<sup>2</sup>*Computer Science Department, School of Informatics, Indiana University*

<sup>3</sup>*Physics Department, Collage of Art and Sciences, Indiana University*

*{byildiz,spallick,gcf}@indiana.edu*

## Abstract

*Web Service is a standardization effort to interoperate loosely-coupled applications. A Web Service interaction benefits and sometimes requires additive functionalities such as security and reliability. They are called as handlers and contribute to build rich, modular and efficient Web Services. However, the way of utilizing them is very crucial for the Web Service Architecture and its overall performance. Using distributed approach for the handler execution facilitates significantly to reach the goal of richness, modularity and efficiency. In this paper we describe an orchestration structure for the distributed handler execution.*

## 1 Introduction

Web Service is defined by W3C as a software system that provides a standard means of interoperating the different software applications, running in a variety of platforms[1]. There are two important nodes in a Web Service interaction: provider and requester. A middleware, which encapsulates a SOAP [2] processing engine and transport helpers, is employed to support the interaction between these nodes. It, called as Web Service container, basically hides the complexity of the SOAP processing and the details of message transportation. It also provides suitable environment for the utilization of additional functionalities such as security, reliability and logging. These additive functionalities are called as handlers. As it is in Apache Axis [3] and Microsoft Web Service Enhancements (WSE) [4], a Web Service container generally uses a processing pipeline to execute them in an order. Although the pipeline allows incrementally adding new functionalities to an interaction, it increases the response time because of many handlers in the execution path. Therefore, we created architecture to efficiently distribute handler to overcome the limitation. We will focus on the orchestration of the handler distribution in this paper. We briefly explain our architecture. We will elaborate the orchestration for

the distributed handlers. Finally, we will provide experimental results and conclude with some remarks.

## 2 Orchestration systems

Many efforts have been spent to obtain a system providing a solution to manage tasks and data in the distributed environments. Academic community joined the effort; GriPhyn[5] provides a good computational environment for the particle physics. SEEK[6] has a solution to orchestrate the tasks for ecology. Taverna [7] offers a flow mechanism for life science. Not only did the academic community provide a solution but there also exist propriety software for the distributed task management such as Inconcert [8], and Websphere MQ Workflow [9]. Moreover, Grid community has an interest in this area because of its focus on secure and collaborative resource sharing across geographically distributed institutions. GridFlow[10] offers an agent-based architecture to schedule the Grid tasks dynamically. Additionally, several new specifications have been presented such as Business Process Language for Web Services (BPEL4WS) [11], and Web Services Choreography Interface (WSCI)[12]. There also exist several systems that utilize markup languages for the orchestration purpose. One of them is eXchangeable Routing Language (XRL) It uses XML base documents for the workflow management [13].

## 3 Distributing Web Service Handlers

A Web Service interaction mostly necessitates additional capabilities such as security, reliability, logging, monitoring, and so on. Many specifications have been also introduced to standardize Web Services. When we look at the capabilities and the product of the standardization efforts, we realize that they are good candidates of being handlers. Unfortunately, this richness of handlers does not always bring happiness. Using several handlers together in an interaction, which is inevitable in many case, can unreasonably increase service response time. In other words, Web Service becomes fat. Fortunately, handler distribution comes to rescue to remove this obstacle.

A Web Service gains several advantages with the handler distribution. First of all, parallel execution can be utilized. Nowadays, even in a simple application, we witness many concurrent tasks. For example, a computer game contains hundreds of concurrent executions. Secondly, Handler distribution allows replication of handlers. This is very beneficial when a handler cannot answer requests. Finally, handler distribution improves reusability; they can be easily reached by many services and clients.

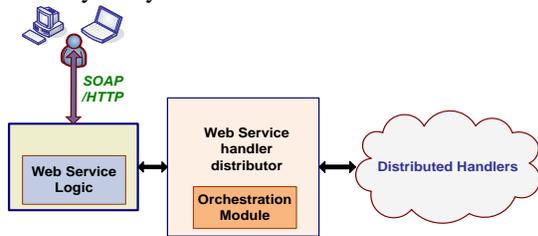


Figure 1 : Distributing Web Service handlers

We created architecture, shown in Figure 1, to benefit from the advantages we have just mentioned. We chose a Message Oriented Middleware (MOM) [14] to distribute the tasks for the handlers. Messaging is one of the key concepts to decouple the distributed applications. Web Services are also familiar with messaging because they are using SOAP messaging over various protocols. The Hypertext Transfer Protocol (HTTP) is the one mostly utilized. It is an application level generic stateless protocol for the distributed collaborative hypermedia information systems [15]. However, HTTP has a limitation because of the request/respond paradigm. The request has to be followed with a response. Therefore, it does not support asynchronous messaging very well. Hence, Utilizing a MOM serves best our purpose. Over this environment, we have introduced an orchestration mechanism. Now, we will elaborate this orchestration.

#### 4 Distributed Handler Orchestration and its XML schema

Orchestration is the key feature of building an efficient distributed execution. There are several approaches; one of them is the use of a Markup language. Petri Net Markup Language (PNML) [16] is an example. Similarly, we chose an XML based document to describe the sequence and the resources for the orchestration. An XML document carries semantic as well as syntax. Its structure, content and semantics are described by XML schema. A schema basically defines the shared vocabularies of the instances of an XML document. Now, we will explain the XML schema of the handler orchestration document.

Handler orchestration schema contains several simple , shown in Table 1, and complex elements to define

execution sequence. Simple elements contribute to build complex schema elements. Name, address, oneway and mustPerform are the elements to define a handler. numberOfLooping, numberOfHandler and condition support to fabricate the execution constructs. The time entity is necessary to monitor the states of handlers. Several time-related variables are required to construct a handler. Start, end and execution times are needed to watch a handler execution.

Table 1: Simple elements in Orchestration Schema

<pre> &lt;!--Element Definitions--&gt; &lt;xs:element name="name" type="xs:string"/&gt; &lt;xs:element name="address" type="xs:string"/&gt; &lt;xs:element name="oneway" type="xs:boolean"/&gt; &lt;xs:element name="mustPerform" type="xs:boolean"/&gt; &lt;xs:element name="condition" type="xs:anyType"/&gt; &lt;xs:element name="numberOfHandler" type="xs:short"/&gt; &lt;xs:element name="numberOfLooping" type="xs:short"/&gt; </pre>
--

Table 2: Handler Definition

<pre> &lt;!--Defines Handler--&gt; &lt;xs:complexType name="handlerType"&gt;   &lt;xs:sequence&gt;     &lt;xs:element ref="name"/&gt;     &lt;xs:element ref="address"/&gt;     &lt;xs:element ref="mustPerform"/&gt;     &lt;xs:element ref="oneway"/&gt;     &lt;xs:element name="time" type="timeType" minOccurs="0" maxOccurs="unbounded"/&gt;   &lt;/xs:sequence&gt; &lt;/xs:complexType&gt; </pre>
--

Handler is the most important entity of the orchestration schema. In other words, it is the keystone of the orchestration. Table 2 defines a handler. It consists of several elements. The name is an identifier to increase readability. The address provides uniqueness for the correct message delivery. We keep track of the time related parameters to collect statistical data and to ensure the message delivery. Several elements are added to improve the performance such as oneway and mustPerform.

Table 3 : The execution constructs

<pre> &lt;xs:element name="executionConstruct"&gt;   &lt;xs:complexType&gt;     &lt;xs:choice&gt;       &lt;xs:element ref="sequential"/&gt;       &lt;xs:element ref="parallel"/&gt;       &lt;xs:element ref="looping"/&gt;       &lt;xs:element ref="conditional"/&gt;     &lt;/xs:choice&gt;     &lt;xs:attribute name="position" type="xs:short" use="required"/&gt;   &lt;/xs:complexType&gt; &lt;/xs:element&gt; </pre>
--

The materials in the universe are composed from the elements defined in the periodic table although their numbers are limited. A written document comprises only letters that are defined in an alphabet. A software language has a small set of basic types to build up a

complex syntax. A processor contains the small set of instructions to execute the complex commands. The same concept is applied to the handler orchestration. We defined four basic constructs, shown in Table 3. They are sequential, parallel, looping and conditional. These basic constructs composes complex execution structures.

The common feature of chemical elements, alphabet, basic types of a language and instruction set of a processor is being well-defined. Hence, the four basic constructs of orchestration needs to be well-defined to build more complex structures correctly. Table 4 shows the definition of sequential execution. It must contain at least one handler. The order of the execution depends on the position of the handlers in the construct.

**Table 4: The sequential execution construct**

```
<xs:element name="sequential">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="handler" maxOccurs="unbounded"/>
      <xs:element ref="numberOfHandler"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The parallel execution, shown Table 5, is more complicated than the sequential one. There exist several types of parallel execution. Synchronous execution forces the orchestration engine to complete every handler execution before starting the next construct. On the other hand, in an asynchronous execution, the next construct may start its executions before the completion of the some handlers in the construct. In order to have parallel execution, there must be at least two handlers.

**Table 5 : The parallel execution construct**

```
<xs:element name="parallel">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="handler" maxOccurs="unbounded"/>
      <xs:element ref="numberOfHandler"/>
      <xs:element ref="typeOfParallelExecution"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

**Table 6 : The looping execution construct**

```
<xs:element name="looping">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="handler"/>
      <xs:element ref="numberOfLooping"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Instead of having multiple appearance of a handler, the number of handler repetition is provided to have a neat document structure. Table 6 shows the schema representation of the looping construct. The quantity of

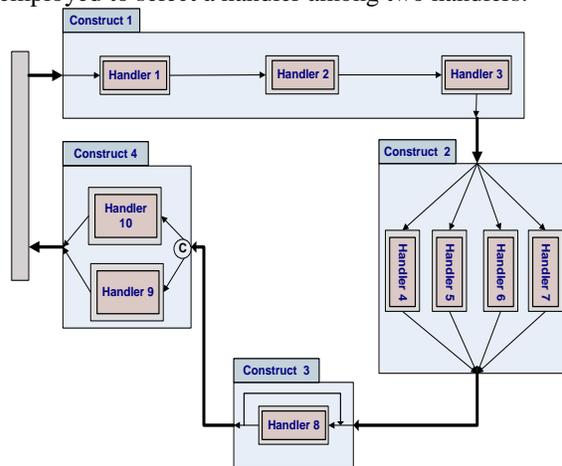
the handlers in a loop is basically one. However, a set of handlers may be processed together many times too. In other words, many handlers can also be in a loop. An execution may decide its sequence according to conditions. We benefited from any type XML element to represent conditions. Table 7 illustrates the construct.

**Table 7 : The conditional execution construct**

```
<xs:element name="conditional">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="handler" axOccurs="unbounded"/>
      <xs:element ref="condition"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

#### 4.1 A handler execution scenario utilizing basic constructs

We create an instance of the orchestration, depicted in Figure 2, to elaborate how to construct a distributed handler orchestration document. We intentionally put a single occurrence from every basic construct. The first construct consists of three handlers running sequentially. The second construct contains four handlers processed concurrently. Each handler starts their executions at the same time while they may complete them in different moments. The third one is a looping construct that many instances of a handler are executed sequentially. Finally, a conditional is employed to select a handler among two handlers.



**Figure 2 : A sample of a handler orchestration**

In sequential construct, the sequence of the execution is defined by the position of the handler in the orchestration document; Handler 1 is followed by Handler 2 and Handler 3 respectively. However, in parallel construct, the order of the handlers is not crucial because the executions start together. In the looping construct, the number of loops describes how many instance of a handler is processed sequentially.

For example, Handler 8 is executed as many times as the parameter defines. Depending on the given condition, the orchestration engine executes either Handler 9 or Handler 10. For example, handler 9 is executed if the SOAP message contains *wsLog* element.

## 4.2 Interpretation of an orchestration document

Conversion of the orchestration structure to the engine understandable execution structure is not in the scope of this paper. However, we want to mention the importance of this concept. The orchestration engine interprets the XML base handler orchestration document, explained above, and creates its internal execution structure to carry out the handler processing. In other words, the constructs in an orchestration document are mapped to the orchestration engine understandable structure. This means the separation of the description from the execution. This notion reduces the complexity of the engine while it is providing a powerful expressiveness. With this decision, the engine that carries out the execution is kept as simple as possible. Simplicity is a significant feature of a software system. Without hurting efficiency, simplicity is the feature being sought in a good design.

## 4.3 Flexibility and policy schema

Although an internal orchestration structure is initially created by utilizing an instance of the orchestration schema, it is possible to alter a sequence while the execution continues. The modification is permissible unless the rules defined are not ignored. An alteration of the internal orchestration structure entails additional controlling mechanisms. Even though the adaptability is an excellent feature so that the system offers a significant flexibility to build a specific execution, necessary policies should be enforced to ensure the correctness of the execution. Some handlers may process any kind of messages arriving to the system without causing any complication. Yet, the others may not be appropriate to be executed without restrictions. There may be a necessity for a compulsory sequence among some handlers. For example, a decryption handler should be processed at the beginning so that the remaining handlers can understand the message content. Therefore, we come up with another XML Schema to define the policies. Policies define conditions to carry out the execution without having problem. We choose any type element to describe policies. Some policies may be optional although some others must be compulsory. The policy may comprise of many ordering elements to force the necessary restrictions. Moreover, it contains the orchestration

schema file name and its version to let the system know where the policies need to be applied.

## 5 Measurements

We have performed extensive series of the measurements illustrating the advantages of distributed handler execution and its orchestration structure. We will provide the benchmark results gathered from a multiprocessor system, Sun Fire V880. It has Solaris 9 Operating System which is equipped with 8 UltraSPARC III processors operating at 1200 MHz with 16 GB Memory. Deployment is made by using Apache Axis 1.2 and Apache Tomcat 5.5.20.

### 5.1 Performance measurements

Distributed handler execution allows utilizing additional resources. There can be many types of resources such as processor, memory, storage or even an application. Although distribution improves the system performance because of the parallelism and additional resources, the management of these components may also cause overhead. Hence, we will investigate the system performance in a multiprocessor system in the remainder of this section.

#### 5.1.1 Handler configurations

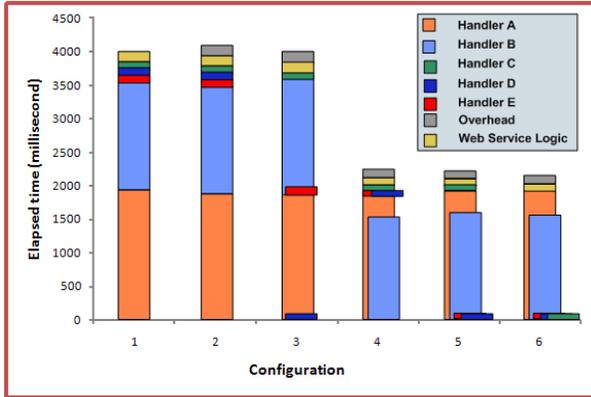
Distributed handler execution is evaluated by utilizing 6 different configurations of 5 Web Service handlers. Handlers are customized for benchmarking purposes. Two of them (*A*, *B*) are CPU bound handlers. The remaining three handlers (*C*, *D*, and *E*) have been chosen from the applications that are gradually switching from CPU bound to I/O bound. *Handler C* and *D* respectively utilizes DOM and SAX parsers. Finally, *Handler E* logs the data and prints out the information about the SOAP message.

Apache Axis describes the handler execution sequence by an XML based WSDD configuration file. It supports only sequential execution. On the other hand, we utilized more flexible approach for the handler deployments. The orchestration document, instead, supports parallel execution as well as sequential one. The combinations of the parallel handlers can create so many different configurations. 6 configurations among them are selected for the experimental purpose. The first configuration which is sequential execution is to gather the results from the Apache Axis. The remainders are various configurations using distributed handler execution structure. The second configuration is the exact one with the Apache Axis sequential execution to evaluate the pure overhead coming from the distribution. The others are to show advantage of using parallelism in handler execution.

#### 5.1.2 The results

The management of the distributed handler execution and the transportation of the tasks affect the execution

time. The cost is inevitable but its burden can be reduced by reshuffling the configuration. In this section, our interest is to find out the performance benefits coming from the advantages of the distribution by using our orchestration mechanism.



**Figure 3 : The service execution times of the six handler configurations containing the five handlers**

The values in Figure 3 show the round trip time of a service request for 6 configurations. Clients record the initial time of the requests and calculate the elapsed time when they receive the responses. Hence, the measurements contain transportation, management of the orchestration and execution times of the service including handlers. Every measurement observed 100 times. Table 8 shows the numerical values of the results and their standard deviations.

**Table 8 : The elapsed time for the service execution and the standard deviation of the performance benchmark**

Configuration number	Mean value (msec)	Standard Deviation
1	4023.02	83.49
2	4052.07	90.52
3	4025.95	92.56
4	2261.08	86.66
5	2250.96	97.11
6	2171.53	86.22

The difference between configuration 1 and 2 is pure overhead originating from the distribution of 5 handlers. The first configuration utilizes Apache Axis handler deployment. The second configuration distributes the handlers to the individual processors. They are both sequential. The remainder configurations show the various parallel executions. Overlapping parts shows the parallelisms. For example, Handler A and D as well as Handler B and E are parallel executions in configuration 3. It is clear that the best results are observed when all handlers run concurrently. However, processing all handlers concurrently may not be always possible.

## 5.2 The overhead originating from the distribution for a single handler

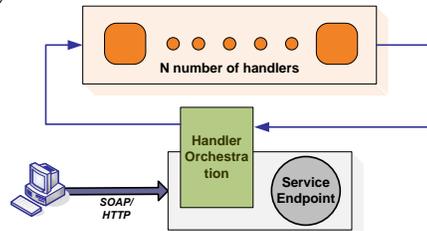
Even though the distribution of handlers provides many advantages to Web Services, it is not free from the cost. Positioning a handler away from Web Service endpoint adds a cost. This cost can be kept in a reasonable range so that the relocation can be justified. In the remainder of this section, we will investigate the overhead for a single handler distribution.

For the sake of the fairness, the results have been gathered by utilizing the same environments with equal parameters. The only difference is the distribution. The parameters carefully selected equal. Measurement starts from 1 handler. The number of the handler is increased by 10 in every step. We continue to add the same handler into the execution path until having 50 handlers. Figure 4 illustrates how the handlers are deployed in Apache Axis.

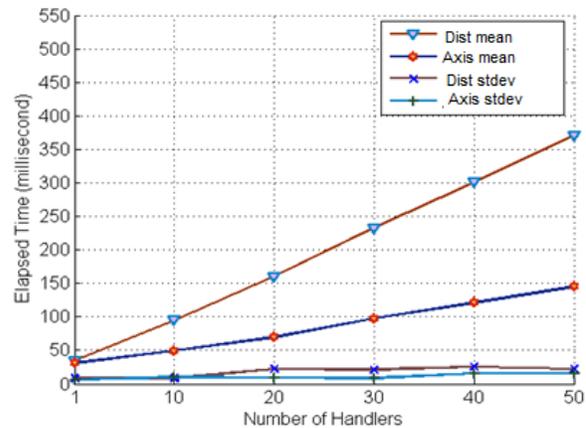


**Figure 4 : Apache Axis sequential handler deployment to measure the overhead**

The same deployment strategy is applied in the distributed approach. Figure 5 illustrates the sequential deployment for the same number of handlers.



**Figure 5 : DHArch sequential handler deployment to measure the overhead**



**Figure 6 : Comparison for the handler addition in Axis 1.x and DHArch**

Every measurement is observed 100 times. At the end, the service elapsed times are collected and the average values are computed. After gathering the values, shown in Figure 6, the overheads, provided in Table 9, are calculated with the following formula:

$$\text{Overhead} = (T_{\text{dist}} - T_{\text{axis}}) / N$$

Where,  $T_{\text{dharch}}$  is the elapsed time of a service utilizing DHArch.  $T_{\text{axis}}$  is the elapsed time of a service utilizing Apache Axis.  $N$  is the number of the handlers in the deployment.

**Table 9 : Overheads of a handler distribution for the increasing number of handlers in the execution path**

Number of handlers	Overhead (msec)
1	4.54
10	4.61
20	4.55
30	4.51
40	4.49
50	4.50

## 6 Future work

The distribution of the handlers puts many choices in front of us. Because of the parallelism, the handler orchestration can be achieved in many ways. However, the throughput cannot be increased by a randomly selected handler sequence. Having an agent that intelligently looks for a better handler orchestration sequence is a very interesting. This agent automates the handler orchestration and adjusts the handler sequence for the best throughput. Hence, finding out the best handler deployment configuration is very promising research area.

## 7 Conclusion

Orchestration is a significant feature to collaborate the distributed applications. Dissemination of the handlers to have efficient and effective SOAP message execution requires a well-organized orchestration. We introduced an orchestration structure separating description from the execution. The separation has many benefits. First of all, it contributes to a very efficient and effective orchestration engine while it is providing very powerful expressiveness in the description. Without sacrificing the efficiency, acquiring simplicity is very appealing.

Secondly, the separation helps us to build static and dynamic handler executions. The orchestration document statically describes handlers and their sequences. It can also create a dynamic handler execution. The execution sequence can be optimized on the fly. The sequence can be altered via introducing parallel execution among the appropriate handlers or rearranging the order. This arrangement must be controlled by policies, which impose the rules to obey the dependencies.

Finally, conventional handler execution mechanism employs a service specific handler sequence. In contrast, we are able to build an individual handler execution sequence for each message by using the introduced orchestration mechanism. This grants significant flexibility that every message may have its specific set of handlers and sequence.

## 8 References

- [1] Web Service Architecture, <http://www.w3.org/TR/ws-arch>.
- [2] Simple Object Access Protocol (SOAP), <http://www.w3.org/TR/soap12-part1>.
- [3] Apache Axis. <http://ws.apache.org/axis>.
- [4] Microsoft Web Service Enhancements (WSE), <http://www.microsoft.com/downloads/details.aspx?FamilyId=FC5F06C5-821F-41D3-A4FE-6C7B56423841&displaylang=en>.
- [5] Ewa Deelman, et al., *GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists* hpdc, p. 225, 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 '02), 2002.
- [6] C. Berkley, et al., *Incorporating semantics in scientific workflow authoring* In Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM'05).
- [7] Oinn, T., et al., *Taverna: lessons in creating a workflow environment for the life sciences* *Concurr. Comput. : Pract. Exper.* 18, 10 (Aug. 2006), 1067-1100
- [8] TIBCO Software Inc Inconcert, <http://www.tibco.com>.
- [9] Aggarwal, B.A., A. Chandra, M. Snir, *A model for hierarchical vmemory*, Proceedings of the nineteenth annual ACM conference on Theory of computing, p.305-314, January 1987, New York, New York, United States.
- [10] Cao, J., et al., *GridFlow: workflow management for grid computing*, Proceedings of 3th International Symposium on Cluster Computing and the Grid, Tokyo, Japan, May 12-15, 2003. IEEE Computer Society Press, 198-205. p. 198-205.
- [11] Curbera F, et al., *Business Process Execution Language for Web Services (BPEL4WS)* <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>.
- [12] Web Service Choreography Interface (WSCI) 1.0 <http://www.w3.org/TR/wsci>.
- [13] Aalst, W.M.P.v.d. and A. Kumar, *XML Based Schema Definition for Support of Inter-organizational Workflow*. University of Colorado and University of Eindhoven report, 2000
- [14] Tran, P., Greenfield, P., and Gorton, I., *Behavior and Performance of Message-Oriented Middleware Systems*. Proceedings of the 22nd international Conference on Distributed Computing Systems, ICDCSW. 2002
- [15] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., *1999 Hypertext Transfer Protocol -- Http/1.1*. RFC. RFC Editor
- [16] J'ungel, M., E. Kindler, and M. Weber. *The Petri Net Markup Language*. Petri Net Newsletter, 59:24-29, 2000