A Grid Workflow Language Using High-Level Petri Nets

Martin Alt, Sergei Gorlatch
{mnalt|gorlatch}@uni-muenster.de
Westfälische Wilhelms-Universität Muenster, Germany
Andreas Hoheisel, Hans-Werner Pohl
{andreas.hoheisel|hans.pohl}@first.fraunhofer.de
Fraunhofer FIRST, Berlin, Germany

CoreGRID Technical Report
Number TR-0000
January 26, 2006

Institute on Grid Information and Monitoring Services

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

A Grid Workflow Language Using High-Level Petri Nets

Martin Alt, Sergei Gorlatch
{mnalt|gorlatch}@uni-muenster.de
Westfälische Wilhelms-Universität Muenster, Germany
Andreas Hoheisel, Hans-Werner Pohl
{andreas.hoheisel|hans.pohl}@first.fraunhofer.de
Fraunhofer FIRST, Berlin, Germany

CoreGRID TR-0000
January 26, 2006

Abstract

One approach to application programming for the Grid is to implement services with often-used functionality on high-performance Grid hosts and provide them to the users located at clients. Complex applications are created by using several services and specifying the workflow between them.

We discuss how the workflow of Grid applications can be described in an intuitive way as a High-Level Petri Net (HLPN), in order to orchestrate and execute distributed applications on the Grid automatically. Petri Nets provide an intuitive graphical workflow description, which is easier to use than script-based descriptions and is much more expressive than directed acyclic graphs (DAG). In addition, the workflow description can be analysed for certain properties such as deadlocks and liveness, using standard algorithms for HLPNs. We propose a platform-independent, XML-based language, called Grid Workflow Description Language (GWorkflowDL), and show how it can be adapted to particular Grid platforms. As two example target platforms, we discuss Java/RMI and the current WSRF standard.

1 Introduction

An approach to Grid programming that receives much attention is the use of remotely accessible services which are implemented on Grid hosts and provide commonly used functionality to applications running on clients. A popular example of such Grid middleware which follows the paradigm of the *Service-Oriented Architecture (SOA)* are the OGSI-compliant Globus Toolkit 3 [5] and the recently specified Web Services Resource Framework (WSRF) standard [4] with several early stage implementations, such as Globus Toolkit 4 and WSRF.net.

Grid applications for service-based systems are usually not based on a single service, but are rather composed of several services working together in an application-specific manner. An application developer has to decide which services offered by the Grid should be used in the application, and he has to specify the data and control flow between them. We will use the term *workflow* to refer to the automation of both – control and data flow – in Grid applications.

In order to simplify Grid programming, it should be possible to describe an application workflow in a simple, intuitive way. Script-based workflow descriptions – e.g. realised by GridAnt [16] and BPEL4WS [2] – explicitly contain a set of specific workflow constructs, such as *sequence* or *while/do*, which are often hard to learn for unskilled users. Purely graph-based workflow descriptions, such as used by Symphony [12] or Condor's DAGman tool [13], have been proposed. Most of these graphical workflow languages are based on Directed Acyclic Graphs (DAGs). Compared to a script-based description, DAGs are easier to use and more intuitive: communications between different services are represented as arcs going from one service to another. However, DAGs offer only a very limited expressiveness, so that it is often hard to describe complex workflows, e.g. loops cannot be expressed directly. HLPNs allow for nondeterministic and deterministic choice, simply by connecting several transitions to the same input place and annotating

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

edges with conditions. Similarly, since DAGs only have a single node type, data flowing through the net cannot be modelled easily. In contrast, Petri Nets make the state of the program execution explicit by tokens flowing through the net.

We propose a Grid Workflow Description Language (GWorkflowDL) [9] based on High-Level Petri Nets (HLPNs). The novelty of our approach is that we do not modify or extend the original HLPN model in order to describe services and control flow: we use the HLPN concept of edge expressions to assign a particular service to a transition, and we use conditions as the control flow mechanism. The resulting workflow description can be analysed for certain properties such as deadlocks and liveness, using standard algorithms developed for HLPNs. The language itself is platform-independent and provides platform-specific language extensions to adapt the generic workflow to a particular Grid platform. In this paper, we present a Java/RMI as well as a WSRF extension.

The workflow language described in this paper is intended to provide a common approach for the whole life-cycle of Grid applications, consisting of the workflow orchestration, scheduling, enactment, execution, and monitoring. The GWorkflowDL is currently the basis for the K-Wf Grid project [11], and the Java Grid of the University of Muenster. The Fraunhofer Resource Grid [6] uses a closely related approach.

The structure of the paper is as follows: In the next section, we present the underlying Grid infrastructure and describe an example application. In Section 3 we give a brief introduction to High-Level Petri Nets and discuss how HLPNs are used to model service-based Grid applications. Section 4 presents the basic features of the Grid Workflow Description Language and the specific extensions for WSRF and Java/RMI platforms. We conclude our paper in the context of related work.

2 Grid Architecture and Application Example

We assume a Grid system architecture as shown in Fig. 1, where application programs are constructed using a set of services which are implemented on remote high-performance hosts. Services are invoked from a client on remote Grid hosts using a remote method invocation mechanism such as Java/RMI or SOAP. We will use the term *service* for any remotely accessible functionality, such as web services or Java/RMI methods, which transforms a number of input parameters into one or several result values.

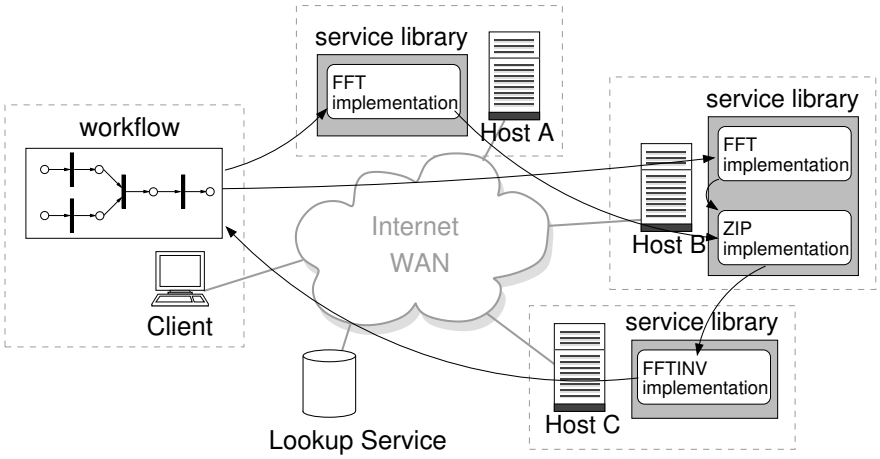

Figure 1: Prototype Grid architecture.

As an application example (also shown in Fig. 1), we use the complex convolution function often used for linear filters in signal processing. The convolution function can be computed efficiently using the Fast Fourier Transform (FFT) and its inverse FFT^{-1} as follows:

$$f_{conv}^{(n,n)}(a,b) = FFT_{2n}^{-1}(FFT_{2n}(a) \times FFT_{2n}(b)) \quad (1)$$

where $\times$ denotes pointwise complex multiplication.

We express the convolution according to (1) using three different services: two services for FFT and FFT^{-1} respectively, and a third service, called zip, for pointwise complex multiplication. Thus, (1) can be specified as

follows:

$$f_{conv}^{(n,n)}(a,b) = FFT^{-1}(zip(FFT(a), FFT(b))) \quad (2)$$

When program (2) is executed on the client, the different services (FFT, zip and FFT^{-1}) are called remotely on (potentially different) Grid hosts, with parameters a and b being sent across the network.

In the remainder of the paper, we develop a workflow for our application example and show how it can be described using HLPNs. We show how the graphical HLPN representation makes parallelism much more explicit than a script-based description and demonstrate how tokens and places reflect the state and the dataflow.

3 Using Petri Nets for Describing Workflows

Our graphical notation for Grid workflow is based on Petri Nets with individual tokens (so-called High-Level Petri Nets), which allow to compute the output tokens of a transition based on the input tokens (as opposed to ordinary Petri Nets where only one type of token exists which is generated by every firing transition). A brief introduction to the theoretical aspects of High-Level Petri Nets can be found, e.g., in Jensen, 1994 [10]. The standardization of the Petri Net concept is currently in progress as an ISO 15909 committee draft [3]. Van der Aalst and Kumar [15] give an overview of how to describe different workflow patterns using Petri Nets.

Petri Nets are directed graphs, containing two distinct sets of nodes: *transitions* (represented by thick vertical lines or rectangles) and *places* (denoted by circles). Places and transitions are connected by directed edges. An edge from a place p to a transition t is called an *incoming edge* of t, and p is called *input place*. Outgoing edges and output places are defined accordingly. Each place can hold a number of individual *tokens* that represent data items flowing through the net. The maximum number of tokens on a place is denoted by its *capacity*. A transition is called *enabled* if there is a token present at each of its input places, and if all ouput places have not reached their capacity. Enabled transitions can *fire* by consuming one token from each of the input places and putting a new token on each of the output places. The number and values of tokens each place holds is specified by the *marking* of the net, which represents the current state of the workflow. Consecutive markings are obtained by firing transitions.

Each edge in the Petri Net can be assigned an *edge expression*. For incoming edges, variable names are used as edge expressions, which assign the token value obtained through this edge to a variable of that name. Additionally, each transition can have a set of boolean *condition* functions. A transition can only fire if all of its conditions evaluate to true for the input tokens.

In the next section, we discuss how individual services can be represented as transitions in High-Level Petri Nets, and Section 3.2 shows how applications built from services are described by more complex nets. We will discuss the workflow concepts using an abstract mathematical notation for edge expressions and show concrete examples for Java/RMI and WSRF in Section 4.2.

3.1 Individual Services

We will illustrate the representation of individual services using the *zip* service for pointwise complex multiplication of two arrays (see Sect. 2). Figure 2 shows the HLPN for the *zip* service.

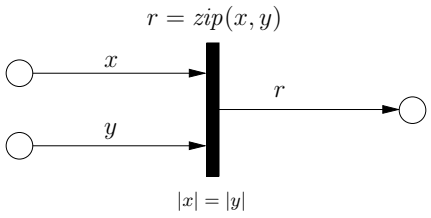

Figure 2: Transition representing the *zip* service.

Services are represented by transitions, with the service name written above the transition. Variables for the formal parameters and results are represented as places, with parameter names (x and y) shown as edge expressions on the incoming edges. The edge expressions for outgoing edges specify which value should be placed on the corresponding output place when the transition fires. Usually, this is the result of the invoked service, but it can also be an error

value. In addition to the service itself, a set of conditions may be associated with a transition. For example, the zip service is only allowed to be executed if the two input lists are of the same length, which is expressed by $|x| = |y|$. The conditions are shown below the corresponding transition.

3.2 Complex Workflows

A larger application is composed of several services, which is expressed graphically by merging output places of one transition with input places of another transition. The resulting network for the convolution example according to equation (2) is shown in Fig. 3. Transitions $load_a$, $load_b$ and $save$ represent local methods used to load the parameter values and save the result.

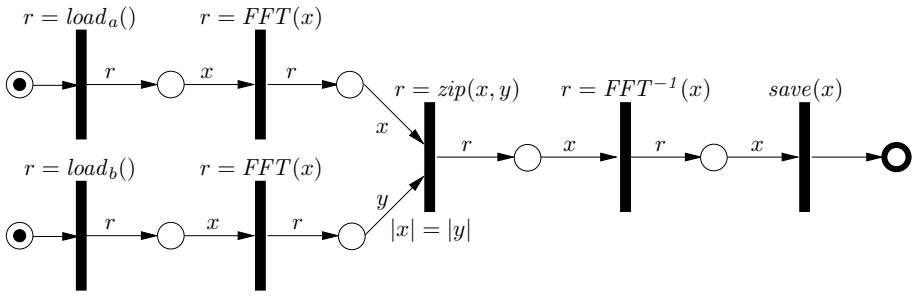

Figure 3: Petri Net representation of convolution program.

3.2.1 Control-Flow

In addition to the places and edges for input and output data of transitions, the application developer has the possibility to introduce additional *control places* to the graph. A control place holds simple tokens which do not carry any value. Accordingly, input edges connecting a control place to a transition (*control edges*) have no associated variables, and the token values are not used as parameters for the associated service. Control edges just synchronize the firing of a transition with the corresponding control place. As example, consider the transitions $load_a$ and $load_b$ in Fig. 3. Because the methods for loading the parameters do not have any input parameters, a control input edge is introduced for each of them. The control places initially contain a single control token which is consumed when parameters are loaded. This ensures that the corresponding transitions are executed only once. be enabled and could fire repeatedly). Similarly, the $save$ transition has an outgoing control edge which signals the completion of the $save$ service.

3.2.2 Conditions

Conditions can be used to check whether the input data of the service meets certain requirements. Additionally, the application programmer can use conditions to realise standard control flow structures, such as conditions and loops. For example, FFT implementations often can only be used on input lists of length $n = 2^i$. This should be checked by a condition provided with the FFT service. However, to use the convolution program in equation (2) for input lists of lengths that are not a power of two, the lists can be zero-padded to the appropriate length.

Figure 4 shows a subgraph for loading list a with padding to the required length. The $padding$ transition calls a service to transform the input list, which is only necessary if the list does not have the correct size. Therefore, the user-defined condition $n \neq 2^i$ is used to prevent unnecessary padding. If the list already has the correct length, it is passed through the second transition which has no associated service and does not change its input.

4 Grid Workflow Description Language

The Grid Workflow Description Language (GWorkflowDL) is being developed as an XML-based language for representing Grid workflows based on HLPNs as described in the previous section. The language consists of two parts: (1) a generic part, used to define the structure of the workflow, reflecting the data and control flow in the application, and

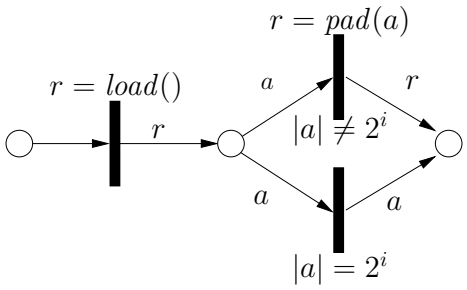

Figure 4: Conditional padding of the input data realised with user-defined conditions.

(2) a platform-specific part (*extension*) that defines how the workflow should be executed in the context of a specific Grid computing platform.

4.1 GWorkflowDL – XML Schema

Figure 5 graphically represents the XML Schema of GWorkflowDL. The root element is called `<workflow>`, which contains the optional element `<description>` with a human-readable description of the workflow, and several occurrences of the elements `<transition>` and `<place>` that define the Petri Net of the workflow.

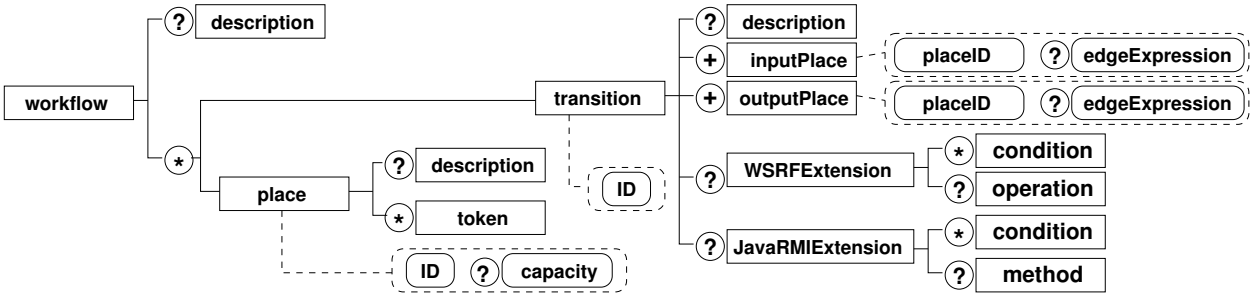

Figure 5: Graphical representation of the XML schema for GWorkflowDL. Boxes denote elements, rounded boxes represent attributes. Legend: $? = 0, 1; * = 0, 1, 2, \ldots; + = 1, 2, 3, \ldots$

The element `<transition>` may be extended by platform specific child elements, such as `<WSRFExtension>` and `<JavaRMIExtension>`, which represent special mappings of transitions onto particular Grid platforms. Elements `<inputPlace>` and `<outputPlace>` define the edges of the net. Edge expressions are represented as attribute `edgeExpression` of `JavaRMIExtension` and `WSRFExtension` tags.

4.2 GWorkflowDL – Platform Extensions

To adapt a generic workflow description to a particular Grid computing platform, we use *extensions*, which describe the meaning of a generic net in the context of a particular platform. We will now present two example extensions, for WSRF and Java/RMI. The platform extensions define: (1) the platform-specific service to be invoked; (2) how conditions are evaluated; (3) how edge expressions are evaluated. As an example, Fig. 6 shows the GWorkflowDL description for the zip service shown in Fig. 2. Note that this code is not intended to be written by the programmer, but instead can either be generated automatically from Java or WSDL interfaces, for single services, or by workflow orchestration tools for combining several services. For WSRF-based applications, each transition represents a web service. Similar to Java/RMI, edge expressions name input parameters used to invoke the web service operation. Fig. 7 shows the corresponding WSRF GWorkflowDL document.

Figure 6 shows the GWorkflowDL representation of the *zip* HLPN (Fig. 2) using the Java/RMI GWorkflowDL extension. The *edge expressions* assign variable names. The *method* element captures services and describes how the

```
<workflow>
  <place ID="P1"/>
  <place ID="P2"/>
  <place ID="R"/>
  <transition ID="ZIP">
    <description>Pointwise complex mult. of two arrays</description>
    <inputPlace placeID="P1" edgeExpression="x"/>
    <inputPlace placeID="P2" edgeExpression="y"/>
    <outputPlace placeID="R" edgeExpression="result"/>
    <JavaRMIExtension>
      <method name="Zip.execute(x,y)"/>
      <condition name="Zip.checkLength(x,y)"/>
    </JavaRMIExtension>
  </transition>
</workflow>
```

Figure 6: Abbreviated GWorkflowDL representation of the zip-HLPN for the Java/RMI platform.

methods and conditions should be called. The *condition* elements provide conditions which contain any Java statement depending on the variables assigned by the input edge expressions and evaluate to a boolean value.

```
<workflow>
  <place ID="P1"/>
  <place ID="P2"/>
  <place ID="R"/>
  <transition ID="ZIP">
    <description>Pointwise complex mult. of two arrays</description>
    <inputPlace placeID="P1" edgeExpression="x"/>
    <inputPlace placeID="P2" edgeExpression="y"/>
    <outputPlace placeID="R"/>
    <WSRFExtension>
      <operation name="zip" owl="gom.kwfgrid.net/zip.xml">
        <WSClassOperation name="zip_xy" owl="gom.kwfgrid.net/zip_xy.xml">
          <WSOperation name="zip_xy@first" owl="first.fhg.de/zip.xml"/>
          <WSOperation name="zip_xy@iisas" owl="savba.sk/zip.xml" selected="true"/>
          <WSOperation name="zip_xy@cyfronet" owl="agh.edu.pl/zip.xml"/>
        </WSClassOperation>
      </operation>
    </WSRFExtension>
  </transition>
</workflow>
```

Figure 7: Abbreviated GWorkflowDL representation for the WSRF platform.

Figure 7 shows the abbreviated GWorkflowDL representation of the *zip* component from Fig. 2, using the WSRF GWorkflowDL extension as used in K-Wf Grid. The *edge expressions* and *conditions* may be specified as XPath queries. The *operation* element captures several levels of abstraction of web service operations: *operation* describes an very abtract operation without any details, *WSClassOperation* specifies a operation on specific class of Web Services described by their interfaces and functionality, and *WSOperation* are the concrete instances of Web Service operations that match the class. The *owl* attribute links to external semantic descriptions.

4.3 Workflow Orchestration and Execution

To design a program for the Grid, the application developer first selects the services required for a particular application and creates an abstract workflow description, such as the net shown in Fig. 3 for the convolution application. The

resulting abstract workflow description can already be analysed for certain properties such as deadlocks and liveness, using standard algorithms for HLPNs (see e.g. [7]).

After selecting an appropriate service-based Grid computing platform, the application developer has to adapt the abstract Petri Net to the particular platform by assigning particular services and platform-specific edge expressions to transitions. E.g., for the Java platform, a Java method of a remote interface is assigned to each transition, and variable names are assigned to the input and output edges.

The resulting specific HLPN for the desired workflow can then be executed on the Grid by assigning an executing host to each service, either manually (to execute the application on a user-selected set of hosts) or more likely automatically, using a scheduling and resource mapping strategy to select hosts.

Execution of the net then starts by selecting an enabled transition. The tokens on the input places are collected and each of the transition's conditions are evaluated. If all conditions yield true, the corresponding service is invoked. The data related to the tokens on the input places is passed to the service as input parameter. Now, depending on the target platform, either the result of the evaluation of the output edge expressions or the result of the service invocation itself is placed as token on the output places. If any condition evaluates to false, the input tokens are returned to their respective input places. Then the next enabled transition is selected. This process continues until each terminal place holds at least one token, or no enabled transitions with true conditions remain.

5 Conclusions

We have presented our approach for expressing Grid application workflows as High-Level Petri Nets and described GWorkflowDL, an XML-based language for Grid workflows. Petri nets are widely used for modelling and analysing business workflows in workflow management systems (e.g. [14]). The use of Petri Nets for Grid workflow has first been proposed as Grid Job Definition Language (GJobDL) [8] for job-based Grid systems, where a Grid application is composed of several *atomic Grid jobs* which are sent to the hosts for execution. The GJobDL language is similar to the GWorkflowDL, however, it uses a modified HLPN where transitions contain input and output *ports* which represent parameters and results, and edges connect places to ports instead of transitions.

In contrast, our GWorkflowDL specifies Grid workflow as a "standard" HLPN (as defined e.g. in [10]), using conditions for control flow mechanisms and edge expressions to assign parameters and results to edges. Adhering more strictly to the standard model of High-Level Petri Nets allows us to make use of standard algorithms for analysing Nets, e.g. for deadlocks and liveness.

The HLPN representation of a workflow serves four main purposes:

1. It is an intuitive graphical description of the program, making communication between services explicit and allowing users to develop programs graphically without having to learn a specific workflow language.

2. Because applications are developed as unmodified HLPNs, the application's HLPN can be used for analysis and formal reasoning based on the results of previous research in High-Level Petri Nets.

3. The same GWorkflowDL description can be used to monitor and inspect running and finished workflows.

4. Because the GWorkflowDL is divided into an abstract and a platform-specific part, it can be used with different service implementations and Grid platforms.

As future work, we plan to implement a set of tools for workflow orchestration, execution, monitoring, and analysis, based on the GWorkflowDL. For example, we intend to implement performance prediction of Grid applications by using time values as tokens and service functions that add the expected performance of particular services (which can be obtained using an approach discussed in [1]) to the input tokens. We also intend to investigate how existing deadlock detection algorithms for HLPNs can be applied to Grid workflows and how to integrate these algorithms into a workflow analysis tool.

Acknowledgements

The work described in this paper is supported in part by the European Union through the IST-2002-004265 Network of Excellence CoreGRID and the IST-2002-511385 project K-WfGrid.

References

[1] M. Alt, H. Bischof, and S. Gorlatch. Program development for computational Grids using skeletons and performance prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, 2003.

[3] Committee Draft ISO/IEC 15909. High-level petri nets – concepts, definitions and graphical notation, Oct 1997. Version 3.4.

[4] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework, May 2004. http://www.globus.org/wsrf/.

[5] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, jun 2002.

[6] Fraunhofer Gesellschaft. Fraunhofer Resource Grid homepage. http://www.fhrg.fraunhofer.de/, 2005.

[7] C. Girault and R. Valk, editors. *Petri Nets for Systems Engineering*. Springer-Verlag, 2003.

[8] A. Hoheisel and U. Der. An XML-based framework for loosely coupled applications on grid environments. In P. Sloot, editor, *ICCS 2003*, number 2657 in Lecture Notes in Computer Science, pages 245–254. Springer-Verlag, 2003.

[9] A. Hoheisel and H.-W.Pohl. Documentation of the Grid Workflow Description Language toolbox. http://fhrg.first.fraunhofer.de/kwfgrid/gworkflowdl/docs/, 2005.

[10] K. Jensen. An introduction to the theoretical aspects of Coloured Petri Nets. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 230–272. Springer-Verlag, 1994.

[11] K-Wf Grid consortium. K-Wf Grid homepage. http://www.kwfgrid.net/, 2005.

[12] M. Lorch. *Symphony – A Java-based Composition and Manipulation Framework for Computational Grids*. PhD thesis, University of Applied Sciences in Albstadt-Sigmaringen, Germany, July 2002.

[13] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[14] W. van der Aalst. The application of Petri Nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[15] W. M. P. van der Aalst and A. Kumar. Xml based schema definition for support of inter-organizational workflow. University of colorado and university of eindhoven report, 2000.

[16] G. von Laszewski, B. Alunkal, K. Amin, S. Hampton, and S. Nijsure. GridAnt – client-side workflow management with Ant. http://www-unix.globus.org/cog/projects/gridant/, July 2002.