User Tools and Languages for Graph-based Grid Workflows

Andreas Hoheisel

Fraunhofer Institute for Computer Architecture and Software Technology (FIRST)
Kekuléstr. 7, D-12489 Berlin, Germany
andreas.hoheisel@first.fraunhofer.de
http://www.first.fraunhofer.de

1 Introduction

One of the main objectives of Grid computing is the abstraction from the hardware infrastructure as well as hiding implementation details of software components from the user. A modern Grid infrastructure should enable the user not only to execute single tasks on specified hardware resources but also to compose and execute complex Grid applications on distributed, heterogeneous and unreliable hardware resources without taking care about lower-level details. With the Grid, a unified infrastructure is becoming available which allows to host computational resources and use them on demand, but also to combine them and organize dataflow between them. For the latter purpose, the concept of *Grid workflow* has emerged which describes patterns of control and dataflow between Grid resources, including – apart from software components and data sources – human actors participating in interactions.

Several techniques have been established in the Grid community in order to define the workflow of Grid jobs. A very promising approach – from the view of the unskilled user – is the usage of graphs for this purpose. While graphs are primarily mathematical abstract entities, they possess very intuitive ways of visualization that can be handled easily even by non-expert users. The main limitation of graphs, however, is the fact that they may become very huge if you use them to model complex workflows. In this case, a hierarchical graph definition that allows graph coarsening and refinement may be a solution. Many Grid workflow approaches build on a special subclass of graphs – the directed acyclic graphs (DAG) – which are easy to implement, but restrict the kinds of workflows that can be modeled.

The aim of this paper is not to give a broad overview about workflow description languages and tools in general but it will rather describe user tools and workflow schemes developed in the Fraunhofer Resource Grid (FhRG) [6] as exemplary solutions. In contrast to other workflow approaches which usually are based on directed acyclic graphs, the FhRG workflow is built on the more expressive formalism of Petri nets. *Dynamic workflow graph refinement* is introduced as a powerful technique to transform abstract workflow graphs into the concrete ones needed for execution and to automatically add fault tolerance to complex workflows.

The Fraunhofer Resource Grid is a Grid initiative of several Fraunhofer institutes funded by the German federal ministry of education and research with the main objective to develop and to implement a stable and robust Grid infrastructure within the Fraunhofer-Gesellschaft, to integrate available resources, and to provide internal and external users with an easy-to-use interface for controlling distributed applications and services in the Grid environment [6]. The component environment supports loosely coupled software components where each software component represents an executable file that reads input files and writes output files. The execution of such a software component we call *atomic job* [8]. We plan to include *Grid Service invocations* as atomic jobs in future releases of the FhRG framework in order to make it OGSA compatible. We will distribute most of the software developed within the Fraunhofer Resource Grid using an Open Source License (GNU GPL) under the label eXeGrid [5].

Fig. 1 depicts the architecture of the Fraunhofer Resource Grid that is currently built on top of the Globus 2.4 toolkit [25]. Two main workflow-related user tools are being developed within that framework: The so-called *Grid Job Builder* provides a graphical user interface to compile an XML-based Grid job description document (GJobDL) [7]. The *Grid Job Handler* parses this document and enacts the Grid job workflow.

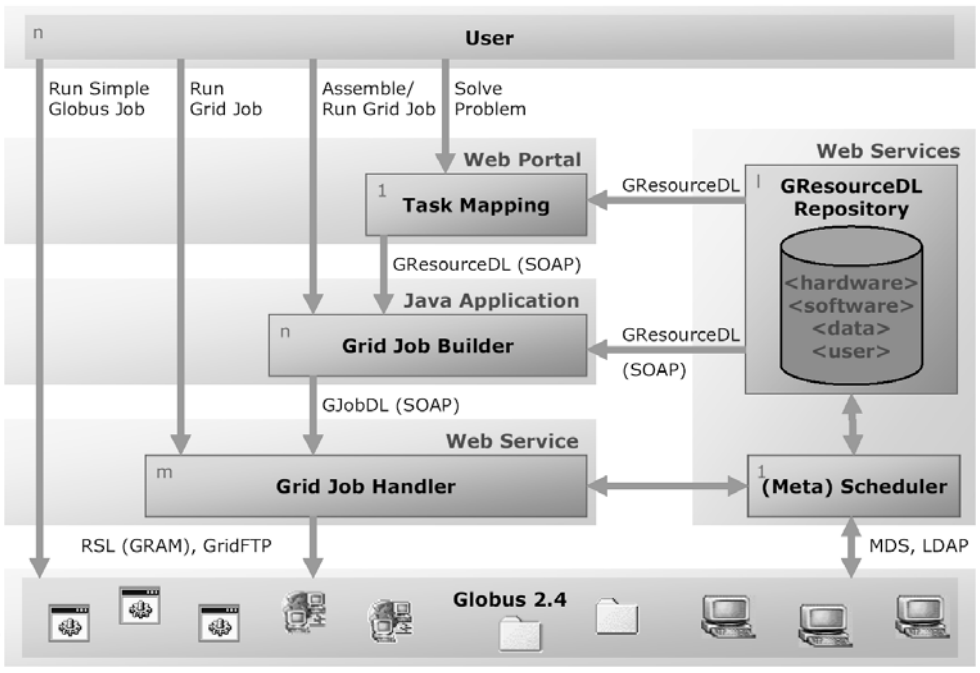
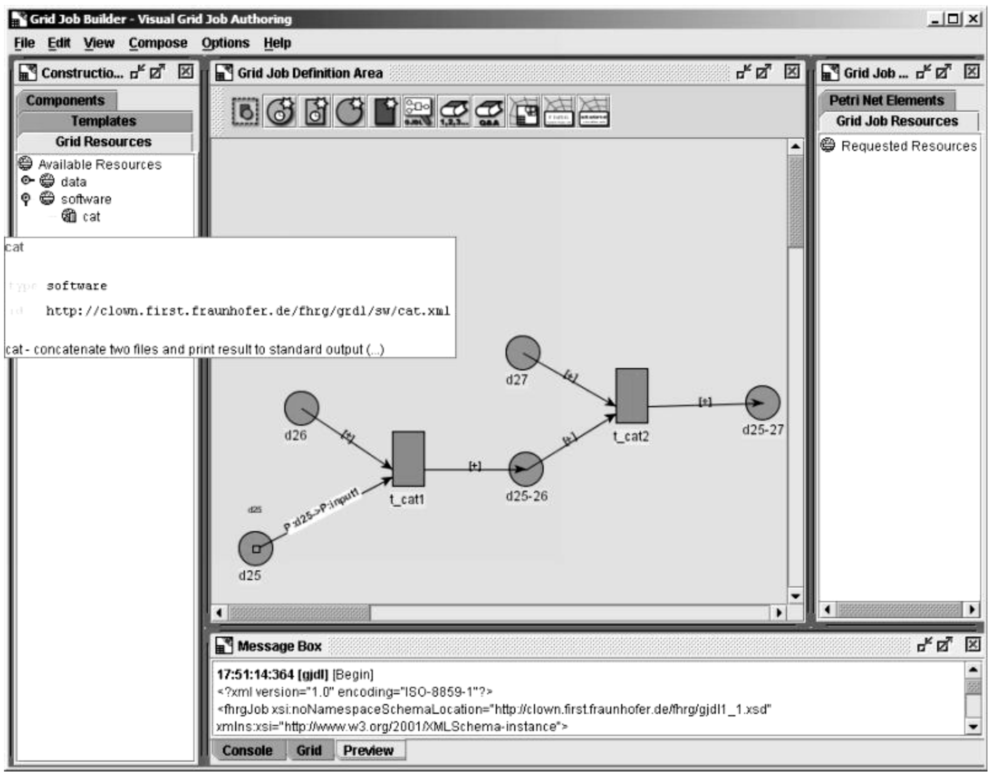


Fig. 1. The layered Grid architecture of the Fraunhofer Resource Grid. The numbers in the upper left corners denote the cardinality

2 Grid Job Orchestration

We define the term *Grid job* as a Grid application that is composed of several Grid resources with a specified workflow. A Grid job may induce a variety of single tasks (atomic jobs) that are indivisible components of a Grid job. According to our definition, Grid resources are either abstract classes or concrete instances of software, hardware or data.

There are several possibilities to provide a workflow management that coordinates the execution of Grid jobs. The workflow is either defined inherently by the software components (respectively Grid Services) themselves, or by software agents that act on behalf of the software components, resulting in a self-organizing or hard-wired Grid job. Another alternative is to define the workflow on a meta level on top of the software components, providing a complete view of the workflow. To describe this kind of workflow it is very important to have suitable semantics.



```
cat

type software
id   http://clown.first.fraunhofer.de/fhrg/grdl/sw/cat.xml

cat - concatenate two files and print result to standard output (...)
```

```
17:51:14:364 [gjdl] [Begin]
<?xml version="1.0" encoding="ISO-8859-1"?>
<fhrgJob xsi:noNamespaceSchemaLocation="http://clown.first.fraunhofer.de/fhrg/gjdl1_1.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Fig. 2. A screenshot of the Grid Job Builder, developed by Fraunhofer IGD. The Grid Job Builder includes a Grid resource browser (*left*), a composition panel for Petri-net-based workflows (*middle*), a job inspector (*right*), and a message box (*bottom*). The Grid Job Builder supports drag and drop to introduce new components to the Grid job workflow

There are three main approaches to achieve the workflow description: It may be based on scripting languages (e.g., GridAnt [15], and JPython in XCAT [13]), on graphs (e.g., Condor DAGman [3], and Symphony [18]), or on a mixture of both (e.g., WSFL [17], XLANG [22], BPEL4WS [2], UNICORE [4], and GSFL [14]). Although the scripting language approaches may be very convenient for skilled users, they are not really intuitive and they are limited by the vocabulary provided by the scripting language as every type of workflow needs other language elements (e.g., for sequential or parallel execution, loops, conditions, etc.).

In our approach, we use a Petri-net-based workflow model that allows the graphical definition of arbitrary workflows with only few basic graph elements – just by connecting data and software components. Fig. 2 shows a screenshot of the Grid Job Builder, a Java application providing a graphical user interface for assembling Grid jobs. The output of the Grid Job Builder is a GJobDL document, which defines the Grid job. This GJobDL document can be saved as a file or transmitted directly to the Grid Job Handler Web Service in order to enact the workflow.

The GJobDL description of a Grid job contains the resource descriptions of the basic resources that are required to define the Grid job and the model of the Grid job workflow using the concept of Petri nets [20]. As mentioned before, many other Grid projects model the workflow using directed acyclic graphs (DAG). One example for this approach is UNICORE, where so-called AbstractJobs are defined on the basis of DAGs [4]. Other prominent Grid projects using DAGs are Condor [3], [21], Cactus [23], and Symphony [18]. DAGs are widely spread due to their simple structure, they possess, however, some relevant disadvantages: DAGs are acyclic, so it is not feasible to explicitly define loops (while...do) without additional language elements that are not related to the graph representation. Furthermore, a DAG only describes the behavior, but not the state of the system. We decided to use Petri nets instead of DAGs (see Fig. 3). Dan C. Marinescu describes a similar approach in his book about internet-based workflow management [19].

```
PARENT A CHILD B C
PARENT B C CHILD D
```

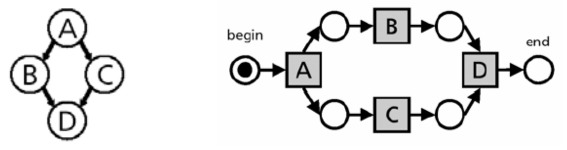


Fig. 3. Example of a directed acyclic graph (*left*) and the equivalent Petri net (*right*)

Petri nets belong to a special class of directed graphs. The type of Petri nets we introduce here corresponds to the concept of Petri nets with individual tokens (colored Petri net) and constant arc expressions which are composed of places, denoted by circles (○), transitions, denoted by boxes (□), arcs from places to transitions (○→□), arcs from transitions to places (□→○), individual and distinguishable objects that flow through the net as tokens (•), an initial marking that defines the objects which each place contains at the beginning, and an expression for every arc that denotes an individual object. A place *p* is called input place (output place) of transition *t* if an arc from *p* to *t* (from *t* to *p*) exists. A brief introduction to the theoretical aspects of colored Petri nets can be found, e.g., in [12]. The standardization of the Petri net concept is currently in progress as an ISO 15909 committee draft [11].

Petri nets possess special mathematical characteristics that are used to analyze and to classify Petri nets. Terms like *conflict*, *confusion*, *contact*, *pit*, and *deadlock* are well-defined properties of Petri nets that may be helpful when analyzing and optimizing the workflow of a Grid job. The actual state of the workflow is represented by the marking of the Petri net. An overview of how to describe different workflow patterns using Petri nets can be found in [1]. Petri nets are suitable to describe the sequential and parallel execution of tasks with or without synchronization; it is possible to define loops and the conditional execution of tasks.

We use Petri nets not only to *model*, but furthermore to *control* the workflow of Grid jobs. In most cases, the workflow within Grid jobs is equivalent to the dataflow, i.e., the decision when to execute a software component is taken by means of availability of the input data. Therefore, the tokens of the Petri net represent real data that is exchanged between the software components or Grid Services. In this case, we use Petri nets to model the interaction between software resources represented by software transitions, and data resources represented by data places. In some cases, however, the workflow is independent from the dataflow, and in addition to the data places and software transitions we have to introduce control places and control transitions. The corresponding tokens contain the state of the process (e.g., done, failed). Control transitions evaluate logical conditions. For further details about the Petri net approach of the Fraunhofer Resource Grid refer to [9], and [10].

There already exist several approaches to describe Petri nets with XML-based description languages. Widely spread is the Petri Net Markup Language (PNML) developed by the Humboldt-Universität zu Berlin [26]. We introduced a dedicated XML syntax, similar to the PNML, in the GJobDL. The job description consists of the declaration of the places, transitions, and arcs that build the Petri net of the Grid job. Transitions and places may be linked to external or internal resource descriptions. Control transitions may possess conditions that are evaluated prior to the firing of activated transitions.

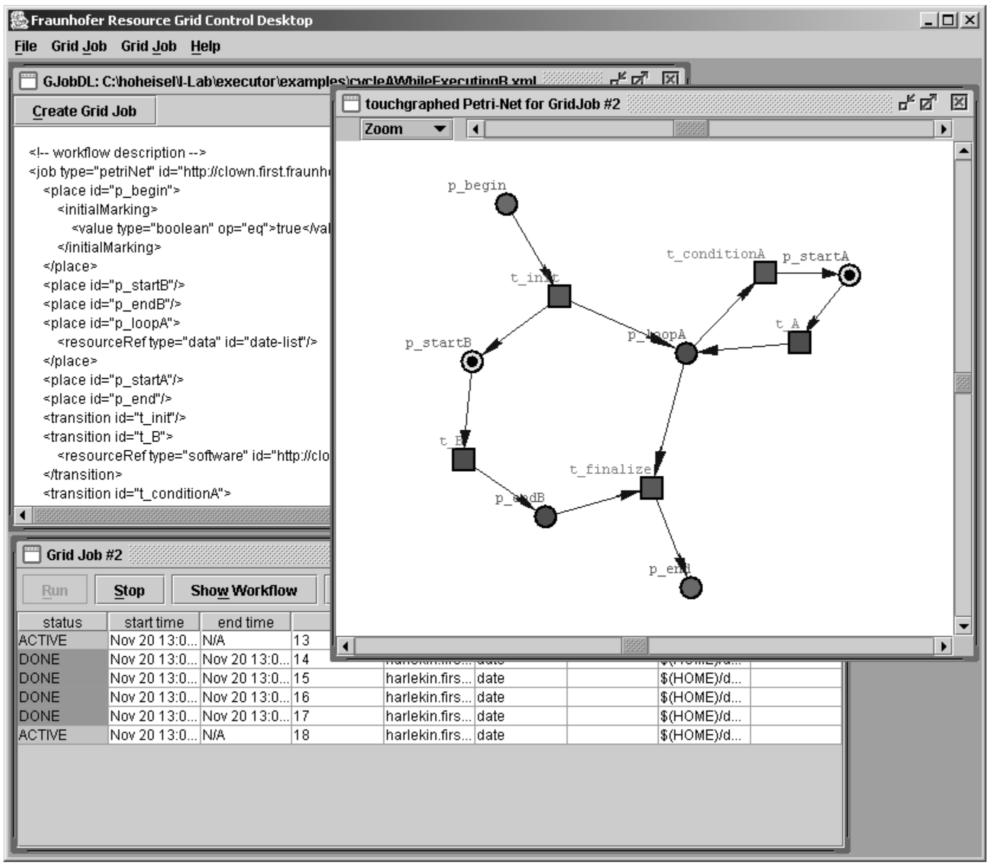
GJobDL excerpt of the Grid job displayed in Fig. 2:

```
<!-- data: d25 -->
<resource id="d25" type="data">
  <location>
    <resourceRef id="gridNode15" type="hardware"/>
    <directory>/home/fhrgdata</directory>
    <filename>d25.dat</filename>
  </location>
</resource>
...

<!-- workflow description -->
<job type="petriNet" id="concatenateIt">
  <place id="d25">
    <resourceRef id="d25" type="data"/>
    <initialMarking>
      <value type="boolean" op="eq">true</value>
    </initialMarking>
  </place>
  ...
  <place id="d25-27">
    <resourceRef id="d25-27" type="data"/>
  </place>
  <transition id="t_cat1">
    <resourceRef type="software" id="cat"/>
  </transition>
  <transition id="t_cat2">
    <resourceRef type="software" id="cat"/>
  </transition>
  <arc id="arc1" type="P2T">
    <placeRef id="d25"/>
    <transitionRef id="t_cat1">
      <inputRef id="input1" type="file"/>
    </transitionRef>
  </arc>
  ...
</job>
```

3 Workflow Enactment

The Grid Job Handler is responsible for the enactment of the Grid job workflow. Therefore, the Grid Job Handler parses the Grid job description, resolves the dependencies between the Grid resources, and searches for sets of hardware resources that fulfill the requirements of each software component. A meta scheduler (see Fig. 1) selects the best-suited hardware resource of each set of matching hardware resources according to a given scheduling policy (fastest, cheapest, etc.). In the current implementation, the Grid Job Handler maps the resulting atomic jobs onto the Globus Resource Specification Language (RSL) [24] and submits them via GRAM to the corresponding Grid nodes. For the communication between the Grid Job Handler and the Globus Grid middleware, we use a patched version of the Java Commodity Grid Kit [16]. The Grid Job Handler itself is deployed as a Web Service with possibilities to create, run and monitor Grid jobs remotely. The desktop version of the Grid Job Handler includes a graphical user interface (see Fig. 4) and additional command line tools.



```
<!-- workflow description -->
<job type="petriNet" id="http://clown.first.fraunh
  <place id="p_begin">
    <initialMarking>
      <value type="boolean" op="eq">true</val
    </initialMarking>
  </place>
  <place id="p_startB"/>
  <place id="p_endB"/>
  <place id="p_loopA">
    <resourceRef type="data" id="date-list"/>
  </place>
  <place id="p_startA"/>
  <place id="p_end"/>
  <transition id="t_init"/>
  <transition id="t_B">
    <resourceRef type="software" id="http://clo
  </transition>
  <transition id="t_conditionA">
```

status	start time	end time						
ACTIVE	Nov 20 13:0...	N/A	13					
DONE	Nov 20 13:0...	Nov 20 13:0...	14	harlekin.firs...	date		$(HOME)/d...	
DONE	Nov 20 13:0...	Nov 20 13:0...	15	harlekin.firs...	date		$(HOME)/d...	
DONE	Nov 20 13:0...	Nov 20 13:0...	16	harlekin.firs...	date		$(HOME)/d...	
DONE	Nov 20 13:0...	Nov 20 13:0...	17	harlekin.firs...	date		$(HOME)/d...	
ACTIVE	Nov 20 13:0...	N/A	18	harlekin.firs...	date		$(HOME)/d...	

Fig. 4. Screenshot of the graphical Grid Job Handler user interface. The upper left panel displays an excerpt of the GJobDL document. The right panel shows a graphical representation of the corresponding Grid job workflow. The lower left panel lists the atomic jobs that are induced by the Grid job with their actual status

The following steps are iteratively invoked in the kernel of the Grid Job Handler:

0. **Verify** the Petri net (well-formedness, liveliness, deadlocks, pits, etc.).
1. Collect all **activated transitions** of the Grid job.
2. Evaluate the **conditions** of the activated transitions.
3. Invoke **method calls** that are referred by the activated transitions (transfer executable, transfer data, unpack, etc.).
4. If a transition references a software component (respectively Grid Service), invoke the **resource mapping** in order to get a set of matching hardware resources.
5. Ask the **meta scheduler** for the best-suited hardware resource to execute the software component out of the set of matching hardware resources.
6. **Refine** the Petri net if necessary (insert additional data transfer, software deployment, or fault management tasks).
7. **Submit** the **atomic jobs** to the hardware resources (or invoke the Grid Service method call) using the Grid middleware (e.g., GRAM).

8. A **transition fires** if the corresponding atomic job is “done” or has “failed”. Remove tokens from input places and put tokens containing information about the exit status of the atomic job to the output places.
9. **Repeat 1-8** until there are no more activated transitions left.

Note that it does not matter how complex the Grid application becomes, the kernel of the Job Handler remains the same for every type of workflow.

3.1 Dynamic Workflow Refinement

The refinement model of the Petri net theory allows substituting parts of a Petri net by new sub Petri nets. The Grid Job Handler takes advantage of this feature and supplements the workflow during runtime by introducing additional tasks that are necessary to complete the Grid job. The user is not required to model every detail of the workflow – he just has to include the essential transitions and places that are related to the software components and the data he wants to include in his Grid job. Additional tasks that have to be invoked due to specific properties of the Grid infrastructure (e.g., network topology) are detected by the Grid Job Handler and considered by automatically introducing additional transitions and places before or during runtime of the Grid job.

In the current version of the Grid Job Handler, data transfer tasks and software deployment tasks are automatically added to the workflow if they are missing in the initial Grid job definition provided by the user. A data transfer task may be introduced to transfer files that are not available on the remote computer (Fig. 5). A software deployment task may be introduced to install software components on a remote computer (Fig. 6). Further Petri net refinements could concern authorization, accounting, billing and fault management tasks (see next section).

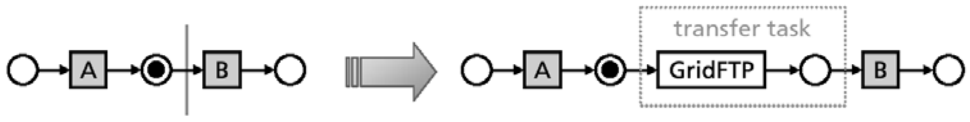


Fig. 5. Software components *A* and *B* are scheduled to different Grid nodes (*left*). A data transfer task

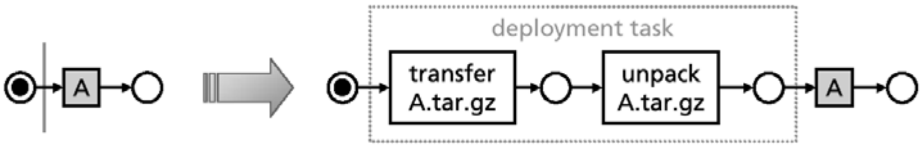


Fig. 6. Software component *A* is to be executed on a Grid node, where it is not yet installed (*left*). A software deployment task is introduced to install the software on the corresponding Grid node (*right*)

3.2 Fault Management

In the Grid computing domain, we distinguish between *implicit* and *explicit* fault management. **Implicit fault management** is inherently included in the Grid middleware and is invoked either by lower-level services regarding fault management of atomic jobs or by higher-level services considering the workflow of the Grid job. This type of implicit fault management can be achieved by Petri net refinement as shown in Fig. 7, where a fault management task is introduced automatically if the submission or execution of an atomic task fails. **Explicit fault management** in our definition refers to user-defined fault management. Within the Petri net workflow model, the user defines the fault management ex-

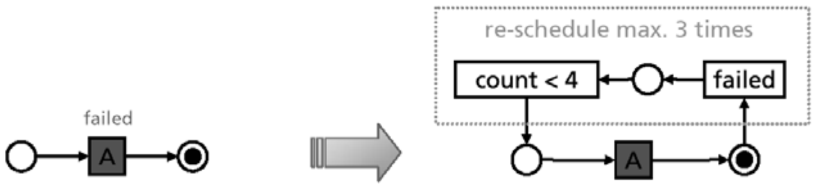


Fig. 7. Example of implicit fault management. If the execution of software component *A* fails (*left*), a fault management task may be introduced into the Petri net (*right*). Here, the fault management task re-schedules the software component maximum three times

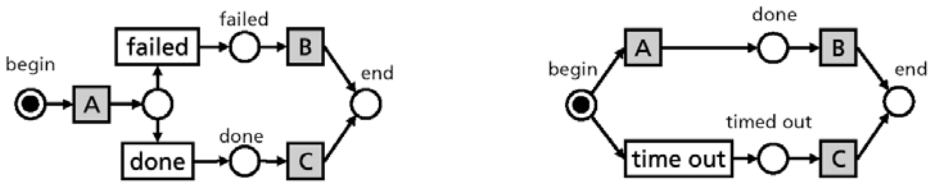


Fig. 8. Two examples of explicit, user-defined fault management. If *A* fails, *B* will be executed; if *A* completes successfully, *C* will be executed (*left*). If *A* does not complete after a specified time out, *C* will be executed. If *A* completes in time, *B* will be executed (*right*)

plicitly by including user-defined fault management tasks in the Petri net of the Grid job. Two examples of user-defined fault management are shown in Fig. 8.

We propose that Grid architectures should provide mechanisms for both, implicit and explicit fault management. The implicit fault management guarantees a basic fault tolerance of the Grid system whereas explicit fault management is needed to support arbitrary, user-defined fault management strategies.

References

1. van der Aalst, W., Kumar, A.: XML based schema definition for support of inter-organizational workflow. (2000)
2. Andrews T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services (BPEL4WS). Specification Version 1.1, Microsoft, BEA, and IBM, ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf (2003)
3. Condor: The Directed Acyclic Graph Manager. http://www.cs.wisc.edu/condor/dagman/ (2003)
4. Erwin, D. W., Snelling, D. F.: UNICORE: A Grid Computing Environment. Lecture Notes in Computer Science, Vol. 2150, Springer-Verlag, Berlin Heidelberg New York (2001) 825–834
5. eXeGrid homepage: http://www.exegrid.net/ (2004)
6. Fraunhofer Resource Grid homepage: http://www.fhrg.fhg.de/ (2004)
7. Fraunhofer Resource Grid: XML schema of the Grid Job Definition Language version 1.1. http://www.fhrg.fhg.de/de/fhrg/schemas/gadl/gjdl.xsd (2003)
8. Hoheisel, A.: Ein Komponentenmodell für Softwarekomponenten des Fraunhofer Resource Grid. Internal report, Fraunhofer FIRST, http://www.andreas-hoheisel.de/docs/FhRGSoftwareComponent.pdf (2002)
9. Hoheisel, A., Der, U.: An XML-based Framework for Loosely Coupled Applications on Grid Environments. In: Sloot, P.M.A. et al. (eds.): ICCS 2003. Lecture Notes in Computer Science, Vol. 2657, Springer-Verlag, Berlin Heidelberg New York http://www.andreas-hoheisel.de/docs/Hoheisel_and_Der_2003_ICCS.pdf (2003) 245–254
10. Hoheisel, A., Der, U.: Dynamic Workflows for Grid Applications. In: Proceedings of the Cracow Grid Workshop '03, Cracow, Poland http://www.andreas-hoheisel.de/docs/Hoheisel_and_Der_2003_CGW03.pdf (2003)
11. ISO 15909: High-level Petri Nets – Concepts, Definitions and Graphical Notation. Committee Draft ISO/IEC 15909, Version 3.4, http://www.daimi.au.dk/PetriNets/standardisation/ (1997)

12. Jensen, K.: An Introduction to the Theoretical Aspects of Coloured Petri Nets. Lecture Notes in Computer Science, Vol. 803, Springer-Verlag, Berlin Heidelberg New York (1994) 230–272
13. Krishnan, S., Bramley, R., Gannon, D., Govindaraju, M., Indurkar, R., Slominski, A., Temko, B., Alameda, J., Alkire, R., Drews, T., Webb, E.: The XCAT Science Portal, SC 2001, ACM SIGARCH / IEEE, Denver (2001)
14. Krishnan, S., Wagstrom, P., von Laszewski, G.: GSFL: A Workflow Framework for Grid Services. Technical Report, The Globus Project, http://www-unix.globus.org/cog/projects/workflow/gsfl-paper.pdf (2002)
15. von Laszewski, G., Amin, K., Alunkal, B., Hampton, S., Nijsure, S.: Gridant – white paper. Technical report, Argonne National Laboratory http://www.globus.org/cog/grant.pdf (2003)
16. von Laszewski, G., Foster, I., Gawor, J., Lane, P.: A Java Commodity Grid Kit. Concurrency and Computation: Practice and Experience 13 (2001) 643–662
17. Leymann, F.: Web Services Flow Language (WSFL 1.0). Technical report. IBM Software group (2001)
18. Lorch, M., Kafura, D.: Symphony — A Java-based Composition and Manipulation Framework for Computational Grids. In: Proceedings of the CCGrid2002. Berlin, Germany (2002)
19. Marinescu, D. C.: Internet-Based Workflow Management – Toward a Semantic Web. Wiley, ISBN 0-471-43962-2 (2002)
20. Petri, C. A.: Kommunikation mit Automaten. Ph.D. dissertation. Bonn (1962)
21. Thain, D., Tannenbaum, T., Livny, M: Condor and the Grid. In: Berman F., Fox, G., Hey T. (eds.): Grid Computing: Making the Global Infrastructure a Reality. John Wiley and Sons Inc (2002)
22. Thatte, S.: XLANG: Web Services for Business Process Design. Specification, Microsoft Corporation (2001)
23. The Cactus Project: http://www.cactuscode.org (2003)
24. The Globus Project: The Globus Resource Specification Language RSL v.1.0. http://www-fp.globus.org/gram/rsl_spec1.html (2000)
25. The Globus Toolkit 2.4. http://www.globus.org/gt2.4/download.html (2003)
26. Weber, M., Kindler, E.: The Petri Net Markup Language. In: Petri Net Technology for Communication Based Systems. Lecture Notes in Computer Science, Advances in Petri Nets, http://www.informatik.hu-berlin.de/top/pnml/ (2002)