Scheduling Workflow Distributed Applications in JavaSymphony*

Alexandru Jugravu[1] and Thomas Fahringer[2]

[1] University of Vienna, Institute for Software Science, Liechtensteinstr. 22, A-1090 Wien, Austria
[2] University of Innsbruck, Institute for Software Science, Technikerstr. 25/7, A-6020 Innsbruck, Austria

Abstract. JavaSymphony is a high-level programming model for performance-oriented distributed and parallel Java applications, which allows the programmer to control parallelism, load balancing, and locality at a high level of abstraction. Recently, we have introduced new features to support the development and the deployment of workflow distributed applications for JavaSymphony. We have built a formal model of a workflow, which allows a graphical representation of the associated workflow. In this paper, we give further details about the workflow model and introduce a new theoretical framework for scheduling JavaSymphony workflow applications.

1 Introduction

Distributed heterogeneous computing has emerged as a cost-effective solution to high-performance computing on expensive parallel machines. In addition, Grid computing has been recently introduced as a worldwide generalization of distributed heterogeneous computing, which has undergone a number of significant changes in a brief time. Supporting grid middleware has expanded significantly from simple batch-processing front-ends to complex tools that provide advanced features like scheduling, reservation and information sharing.

Many complex distributed applications are today structured as workflows that consist of off-the-shelf software components, which are usually applications to be run on individual sequential or parallel machines. The specification and management of workflows is complex and currently the subject of many research projects. Typically, much of the existing work focuses on workflow languages which describe component interconnection features, on the architecture of the enactment engine which coordinates the workflow execution, or on the optimization of the execution by using complex mapping and scheduling techniques.

JavaSymphony is a programming paradigm for wide classes of heterogeneous systems that allows the programmer to control the locality, parallelism, and load balancing at a high level of abstraction without dealing with error-prone and low-level middleware

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 272–281, 2005.

details, like creating and handling remote proxies for Java/RMI or socket communication. The JavaSymphony middleware consists of distributed objects and remote method invocations that run on distributed computing resources like workstation networks and SMP clusters. Moreover, JavaSymphony offers high level features [1] like migration, a distributed event mechanism, and distributed synchronization mechanisms, which are highly useful for developing distributed applications.

Recently, we have built new features on top of the JavaSymphony programming paradigm and runtime system, to support the widely popular workflow paradigm, which include a high-level tool that allows the graphical composition of a workflow, an expressive, yet simple workflow specification language, and an automatic scheduler and enactment engine for workflow applications. In previous work [2], we have presented a formal workflow model consisting of basic elements like activities, control and data flow links, loops, and branches. In this paper, we introduce a theoretical framework for scheduling and propose a scheduling technique for dynamic workflow applications with loops and conditional branches.

The paper is organized as follows. The next section discusses the elements of the workflow model used in JavaSymphony. Section 3 describes the framework for scheduling JavaSymphony workflow applications. Section 4 discusses related work. Finally, some concluding remarks are made and the future work is outlined in Section 5.

2 Workflow Model

A workflow consists of several interconnected computing activities. Between two computing activities there may be: (1) a control flow dependency, which means that one activity cannot start before its predecessors finished or (2) a data dependency, which means that one activity needs input data that is produced by the other. We use the terminology and specifications proposed by the Workflow Management Coalition[3] to define the workflow model and its elements. A graphical representation based on the UML Activity diagram ([4]) is associated with each workflow. In JavaSymphony workflow model, each workflow application is associated with a workflow graph defined by: $WF = (Nodes, CEdges, DEdges, Loops, PLoops, istate, fstate)$.

$Nodes = Act \cup DAct \cup Init \cup Final \cup Branches$ comprises the vertices of the graph associated with 5 types of workflow basic elements: activities, dummy activities, initial states, final states and branches. There are 4 types of edges for the workflow graph: $CEdges$, $DEdges$, $Loops$, respectively $PLoops$ are the sets of the control links, data links, respectively loops and parallel loops of the graph. These basic workflow elements are shortly explained below.

Activities are represented as elements of the Act set. The workflow activities are placed onto computing resources and perform specific computation.

Dummy activities are represented as elements of the $DAct$ set. As a special type of activities, they evaluate complex conditional expressions that may influence the workflow schedule. On the other hand, they require only minimal computing power and therefore they run locally within the scheduler, instead of being placed onto distributed computing resources.

Control links correspond to the elements of the $CEdges$ set. A control-link between two activities means that the second activity cannot start before the first one

finishes. The **control-precedence relation**, denoted by $<$, is defined over the elements of the $Nodes$ set, as the transitive closure of $CEdges$.

Data links define the **data-precedence relation** (denoted by $<_d$) over the set of the activities of a workflow. A data-link between two activities means that the second activity requires output data from the first one.

Initial and final states correspond to the elements in the sets $Init$, respectively $Final$. Each workflow has one entry and one exit point, which we call **initial state**, respectively **final state**. These are used for synchronization of activities and to mark the body of the so-called sub-workflows. They are not associated with computation.

A sub-workflow unit is delimited by a unique pair of an initial state (entry point) and a final state (exit point): $(i, f) \in Init \times Final$.

Conditional branches are represented as elements of the $Branches$ set. Due to the conditional branches, the execution plan of a workflow changes dynamically. The successors of the conditional branch correspond to the entry points (i.e. initial states) of sub-workflows. Each conditional branch exit (control link) is associated with a Boolean expression. When the execution reaches the conditional branch, the Boolean expressions are evaluated and the successors for which this expression evaluates to *false* will not be executed.

(Sequential) Loops are represented as the elements of $Loops \subset Final \times Init$ and may be attached only to entire (sub) workflow units. The body of a (sub)workflow which has a loop associated with it, is executed repeatedly for a fixed number of times (for-loops), or until an associated condition is satisfied (until-loops).

Parallel Loops are represented as the elements of $PLoops \subset Final \times Init$. They are similar with the regular loops, but model a different behaviour of the associated sub-workflow: For each parallel loop, the number of iterations n is specified, and n identical copies of the associated sub-workflow will be created and executed **in parallel**. A parallel loop can be replaced with n identical copies of the associated sub-workflow, but in this case a significantly more complex workflow graph is necessary.

3 Scheduling Workflow Applications

To build a JavaSymphony workflow application, one has to first design the workflow graph, by using the specialized graphical user interface. The developer puts together workflow activities, dummy activities, initial and final states, and connects them using control links, data links, loops and parallel loops, according to the model described in the Section 2. The result is an easy-to-understand workflow graphical representation, based on the UML Activity Diagram, which can be stored in a file by using the specific XML-based specification language. Behind the graphical representation, each element (vertices and edges of the graph) is associated with relevant workflow information. Within the same scheduling process, the workflow specification is analyzed, a resource broker determines which resources are suitable for each workflow activity, a scheduler computes the workflow execution plan, and a enactment engine manages the execution of the activities according to the execution plan. In this section, we present a theoretical framework to describe the scheduling process, and propose a scheduling technique for workflows with branches and loops.

3.1 Scheduling Workflows Without Branches and Loops

We consider first the case of scheduling workflows with no loops and branches. The graph associated with a workflow with no loops and branches becomes a static DAG. Therefore, we call such workflows DAG-based workflows. Scheduling DAGs of tasks is a problem that has been intensively studied, and consequently we can easily use one of the many already existing algorithms [5–8] for scheduling DAG-based workflows. In this section, we introduce several basic definitions and notations related to the scheduling of DAG-based workflows.

If WF is a workflow with $Loops = PLoops = \emptyset$ and $Branches = \emptyset$, then a **schedule** for WF would be a function $\mathbf{sched} : Act \cup DAct \rightarrow M \times \mathbb{R}_+$, where M is the set of computing resources and $\mathbb{R}_+$ is the set of positive real numbers. $\mathbf{sched(T)} = (m_T, start_T)$ means that the activity T is started on machine m_T at the time $start_T$.

The execution time of an activity T on machine m is denoted by $\mathbf{exec(T/m)}$. We assume that the task runs exclusively on that machine. **The communication time** to send data from activity T_1 running on m_1 to activity T_2 running on m_2 is denoted by $\mathbf{comm(T_1/m_1, T_2/m_2)}$. Note that if $T \in DAct$, we may assume m_T is always a dedicated or local machine m_0 (where the scheduler is running) and we consider $exec(T/m_T)$ to be 0. We also assume that communication time for two activities running on the same machine is 0: $comm(T_1/m, T_2/m) = 0$

For a DAG-based workflow WF, a schedule $sched$ is constrained by the workflow control- and data-dependencies:

$T_1 < T_2$ implies $start_{T_1} + exec(T_1/m_{T_1}) \leq start_{T_2}$

$T_1 <_d T_2$ implies $start_{T_1} + exec(T_1/m_{T_1}) + comm(T_1/m_{T_1}, T_2/m_{T_2}) \leq start_{T_2}$

The goal of the scheduler is to find a schedule for each workflow application, which optimize a specific performance function, under certain constraints. Such functions are: makespan (execution time of the whole workflow application), total cost of the resources (when the resources are associated with computation/communication cost) or the throughput of the entire system.

3.2 Scheduling Workflows with Branches and Loops

The conditional branches and the loops in the workflow model enforce dynamic changes in the structure of the execution task graph associated with the application. Subsets of the activities which make up the application may be executed repeatedly several times or may not be executed at all, based on data that is available only at runtime. Consequently, scheduling techniques for static DAG-based workflows cannot be applied in this case.

Our strategy is to transform the workflow associated with the application into one with no conditional branches and loops and recursively find a schedule in the conditions of Section 3.1.

We first define two types of activities: **Unsettled activities** are the activities for which the scheduling/execution decision is taken based on data that is not (yet) available. Such activities are, for example, the activities subsequent to a conditional branch, for which the associated condition cannot be evaluated, because the parameters in the Boolean expression have not been calculated yet. Therefore, it is not sure at this point that these activities will ever be scheduled for execution. The rest of the activities are

called **settled activities**. These are the activities that are planned for execution or have been executed at a specific time of the scheduling/execution process. All the activities for which it is sure that they will be scheduled for execution are considered **settled**. The two sets of activities of a workflow application are dynamically changing during execution, according to the following transformations:

Parallel loop elimination is performed before the scheduling actually starts if the number of the iterations is determined at design time. Otherwise, if the number of iterations depends on the value of workflow relevant data (e.g. variables values), the transformation is applied upon reaching the loop entry (i.e. associated initial state). The body of the parallel loop construct (i.e. the associated sub-workflow) is simply replaced with n identical copies (see Fig. 4(c)).

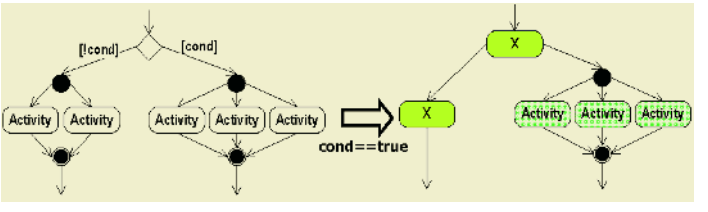


Fig. 1. Branch elimination

Branch elimination is applied when the conditions for the conditional branches are evaluated. This transformation takes place at runtime and is illustrated in Fig. 1. Note that the successors of a conditional branch are unsettled activities (uncoloured in the picture) before the evaluation of the condition, and become settled activities (coloured in the picture) after that. The branches for which the associated condition evaluates to $false$ are not executed. They are replaced by dummy activities (marked as X in the figure), which do not perform any computation.

Transformation of for-loops. The for-loops have a fixed number of iterations. This transformation may take place anytime during the scheduling process. For each iterations of the loop, clones of the activities (i.e. new activities with the same properties as the original ones) in the body of the loop and associated control/data links are added to the graph. The new activity clones preserve the settled state, if the original activities have been settled activities before the transformation.

Transformation of until-loops. The until-loops terminate when a specific condition is fulfilled. The evaluation of the condition can be performed only at runtime. This transformation is illustrated in Fig. 2. For each iteration of the loop, clones of the activities in the body of the loop and associated control/data links are added to the graph. The activities in the first iteration remain settled after the transformation if they have been settled, but the clone activities in the consequent iterations are unsettled. Any activity subsequent to an until-loop preserves its unsettled state until all the iterations of the loop are executed.

Elimination of initial and final states. The initial and final states are simply replaced by dummy activities, not associated with computation. If all their (direct) predecessors are settled activities, these become settled dummy activities.

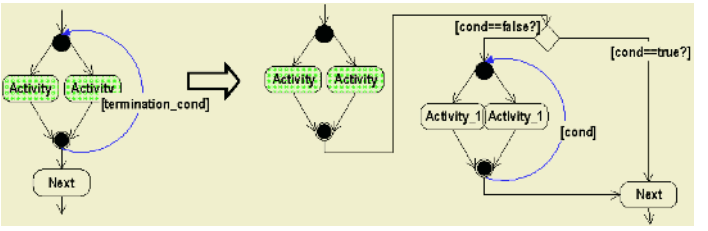


Fig. 2. Until-loops transformation

We use the notation $WF \longmapsto WF_t$ to express that WF_t is obtained from WF applying the above-mentioned transformations. We iteratively build a transformed workflow as follows: Initially (pre-scheduling), all possible transformations, except branch elimination, are applied. The workflow application is scheduled/executed until a conditional branch is reached (i.e. all predecessors of a conditional branch finished their execution). Upon this event the branch elimination is applied, followed by all the other possible transformations. The sets of settled, respectively unsettled activities are recalculated after each transformation step as following.

For $B \in Branches$ a branch node, we denote by $Next(B)$ the set of direct successors of B, which comprises all activities directly dependent via control edges on B and all the activities of the sub-workflows directly dependent via control edges on B. According to this definition, $Next(B)$ comprises all the activities that may be cancelled after reaching the conditional branch B. Note that the decision to cancel or not an activity from $Next(B)$ set can be taken only when the execution reaches B and all conditions associated with the subsequent branches are evaluated.

Consequently, the **set of unsettled activities** is $U(WF_t) = U_1 \cup U_2$, where $U_1 = \bigcup_{B \in Branches} Next(B)$ and $U_2 = \{N \in Act \cup DAct | \exists M \in U_1, M < N\}$. The **set of settled activities** is therefore $S(WF_t) = Act \cup DAct - U(WF_t)$. We denote by $DAG(WF_t) = (S(WF_t), (Edges(WF_t) \cup Loops(WF_t)) \cap S(WF_t) \times S(WF_t))$, the graph which has $S(WF_t)$ as vertices, and all the control links, and loops from WF_t that have both the targets and sources in $S(WF_t)$ as edges.

For a workflow WF, we define **a control path** as a series of activities $A_1, A_2, ...A_k$, where each pair (A_i, A_{i+1}) is either a control link or a sequential loop. Using the above-mentioned notations and definitions, we demonstrate the following property of $DAG(WF_t)$:

Lemma 1. *$DAG(WF_t)$ is a DAG which preserves the control paths of the initial workflow WF.*

Proof:

$DAG(WF_t)$ **has no loops.** According to the transformation of while loops, the body of a loop in WF_t has only unsettled activities. Therefore, the final state associated with a loop is not in $DAG(WF_t)$ and accordingly, the loop is not edge in $DAG(WF_t)$.

$DAG(WF_t)$ **preserves the control paths of** WF means that for each control path $A_1, A_2, ...A_k$ of WF, with all A_i in $S(WF_t)$, there is a corresponding control path in $DAG(WF_t)$. First, the control edges of the initial workflow are preserved by all transformations, so if A_i,$A_{i+1} \in S(WF_t)$ and $(A_i, A_{i+1}) \in CEdges$, implies (A_i, A_{i+1})

is also edge in $DAG(WF_t)$. On the other hand, if (A_i, A_{i+1}) is a for-loop, this means that a for-loop transformation has been applied, followed by an elimination of initial and final states. In this case the loop is transformed into a control link between A_i and a clone of A_{i+1}, both dummy activities in WF_t. If (A_i, A_{i+1}) is an until-loop, this means that a until-loop transformation has been applied, followed by a branch elimination and then by an elimination of initial and final states. In this case the loop is transformed into 2 control links: (A_i, B) and B, A'_{i+1}, where B is a new branch and A'_{i+1} is a clone of A_{i+1} in WF_t and all of them are (newly created) dummy activities.

1. Apply all possible transformations to the initial workflow $WF \longmapsto WF_t$, and compute $U(WF_t)$, $S(WF_t)$ and $DAG(WF_t)$.
2. A scheduling algorithm for DAG-based workflows (no conditional branches and loops) is applied to $DAG(WF_t)$.
3. At each scheduling event, $U(WF_t)$, $S(WF_t)$ and $DAG(WF_t)$ are recalculated. Note that termination of activities may imply adding their successors to $S(WF_t)$. Changes in $DAG(WF_t)$ automatically imply scheduling/rescheduling of unfinished activities.
4. When the execution reaches a conditional branch a branch elimination transformation is applied, followed by all the other possible transformations.
5. The result is a new WF_t, and new $U(WF_t)$, $S(WF_t)$ and $DAG(WF_t)$ are calculated. The scheduling algorithm is now applied to the new $DAG(WF_t)$.
6. The iterative scheduling/execution process finishes when all activities (in all iterations of all loops) are processed. At this point $U(WF_t) = \emptyset$, and $S(WF_t)$ comprises all the activities of WF, including the new created clones of activities (for each additional iteration of a loop) and all new created dummy activities.

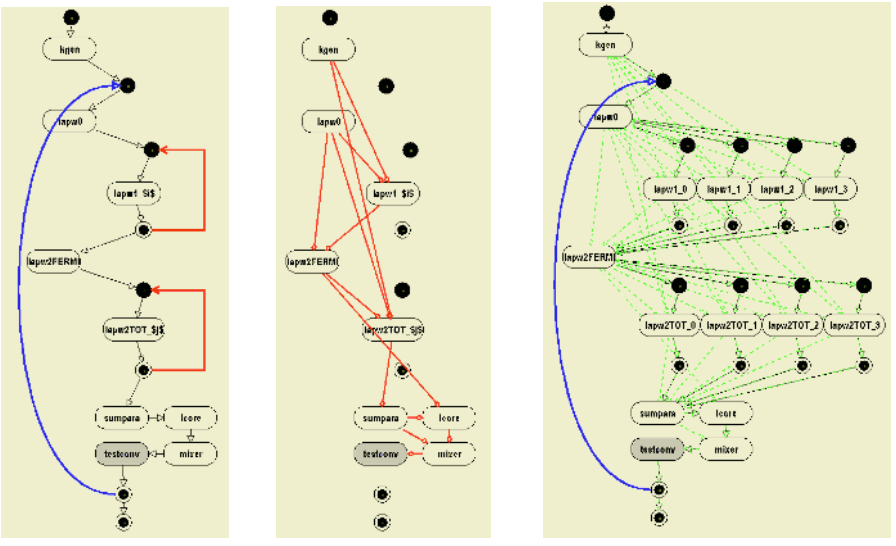
Fig. 3. Strategy for scheduling workflows with loops and branches

Consequently, the dynamic scheduling strategy in Fig. 3 is adopted for workflows with conditional branches and loops.

3.3 A Sample Workflow Application

We have tested our dynamic scheduling strategy with a real-life application. WIEN2k [9] is a program package for performing structure calculations of solids using density functional theory, based on the full-potential (linearised) augmented plane-wave ((L)APW) and local orbitals (lo) method.

The components of the WIEN2k package can be organized as a workflow (Fig. 4). The *lapw1* and *lapw2_TOT* tasks can be solved in parallel by a fixed number of so-called *k-points*. This is modelled by two parallel loops in the workflow graph. Without the parallel loops, the workflow graph becomes quite complex (Fig. 4(c)). Various files are sent from one workflow activity to another, which determine complex data dependencies between the activities (Fig. 4(b)). At the end of the main sequence of the activities, a dummy activity *testconv* performs a convergence test to determine if the calculation needs to be repeated. This is modelled by the main sequential loop.



(a) Control flow (b) Data dependecies (c) Parallel loop elimination

Fig. 4. Wien2k workflow

We have successfully built a JavaSymphony workflow application on top of the WIEN2k package. We have used HEFT (Heterogeneous Earliest Finish Time) [5] list scheduling algorithm combined with the dynamic scheduling strategy described in Fig. 3, to schedule and run this application onto a set of workstations. Due to space limitations, in this paper we do not investigate the workflow scheduling performance. We intend to implement several other real-life distributed applications and to investigate several other scheduling algorithms in future work.

4 Related Work

Workflow applications have become very popular in Grid community and many research and industry groups have proposed language standards to model and develop workflow applications [10–13]. We do not intend to compete with highly complex workflow definition languages [10, 13]. Instead, the JavaSymphony specific XML-based specification language for workflow applications is simple, in order to allow an easy manipulation of the workflow structure by a scheduler. The same is valid for the workflow graphical representation. Activity Diagrams or Petri Nets have been extensively studied as alternatives for the representation of the workflows ([14, 15]). In [15] diverse workflow patterns are analyzed. However complex workflow specification languages or complex workflow patters are not commonly associated with advanced scheduling techniques for distributed workflow applications. We prefer to use a simplified graphical workflow application representation (a reduced set of workflow patterns), in order to be able to investigate such advanced scheduling techniques.

On the other hand, most systems for allocating tasks on grids, (e.g. DAGMan [12], Pegasus [16]), currently allocate each task individually at the time it is ready to run, without aiming to globally optimise the workflow schedule. In addition, they assume that workflow applications have a static DAG-based graph, which may be seen as a too restrictive constraint.

The DAG scheduling problem has been intensively studied in the past, mostly in connection with parallel application compiling techniques. A parallel application is represented by a DAG in which nodes represent application tasks (computation) and edges represent inter-task data dependencies (communication). Numerous scheduling techniques and scheduling heuristics have been developed for both homogeneous and heterogeneous systems [5–8]. However, these heuristics assume a static application graph and they statically compute the schedule before the execution is started. Static scheduling of static DAG structures is, however, too restrictive for the new generation of Grid workflow applications. We, therefore, propose a new approach that includes loops and conditional branches to the workflow model and extends the static scheduling with novel dynamic scheduling techniques to accommodate these new constructs.

5 Conclusions and Future Work

JavaSymphony is a system designed to simplify the development of parallel and distributed Java applications on heterogeneous computing resources ranging from small-scale clusters to large scale Grid systems.

In this paper, we have presented a formal model to describe workflow applications, which allows a user-friendly graphical workflow representation based on the UML Activity Diagram, and a novel framework for scheduling workflow applications.

JavaSymphony introduces a mechanism to control loops and conditional branches in workflow applications, which is not supported by many other workflow frameworks. Furthermore, we describe a new scheduling technique for workflows which have loops and conditional branches.

We plan to evaluate this technique with several DAG-scheduling heuristics [5–8], and compare their performance with several workflow applications. We also plan to further investigate new scheduling techniques for various types of distributed applications and programming paradigms (e.g. meta-tasks, master/slave applications, etc..) and support them in JavaSymphony.

References

1. Jugravu, A., Fahringer, T.: JavaSymphony: A new programming paradigm to control and to synchronize locality,parallelism, and load balancing for parallel and distributed computing. Concurency and Computation, Practice and Experience (2003)
2. Jugravu, A., Fahringer, T.: JavaSymphony, A Programming Model for the Grid. Future Generation Computer Systems (FGCS) **21** (2005) 239–246
3. WfMC: Workflow Management Coalition: http://www.wfmc.org/ (2003)
4. Dumas, M., Hofstede, A.: UML Activity Diagrams as a Workflow Specification Language. In: 4th International Conference on UML, LNCS 2185, Toronto, Canada, Springer Verlag (2001)

5. Topcuoglu, H., Hariri, S., Wu, M.Y.: Task scheduling algorithms for heterogeneous processors. In: Eighth Heterogeneous Computing Workshop, IEEE C.S. Press (1999) 3–14
6. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing **59** (1999) 381–422
7. Baskiyar, S., SaiRanga, P.C.: Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length. In: Proc. of International Conference on Parallel Processing Workshops,Kaohsiung, Taiwan. (2003)
8. Radulescu, A., van Gemund, A.J.C.: Fast and effective task scheduling in heterogeneous systems. In: Heterogeneous Computing Workshop. (2000) 229–238
9. P.Blaha, K.Schwarz, G.Madsen, D.Kvasnicka, J.Luitz: WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties. Vienna University of Technology (2001)
10. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Systems, S., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services (bpel4ws). Specification version 1.1, Microsoft, BEA, and IBM (2003)
11. Erwin, D.W., Snelling, D.F.: UNICORE: A Grid computing environment. Lecture Notes in Computer Science **2150** (2001) 825–??
12. The Condor Team: Dagman (directed acyclic graph manager) (2003) http://www.cs.wisc.edu/condor/dagman/.
13. Krishnan, S., Wagstrom, P., von Laszewski, G.: GSFL : A Workflow Framework for Grid Services. Technical Report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A. (2002)
14. Eshuis, R., Wieringa, R.: Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. Lecture Notes in Computer Science **2472** (2003) 321–351
15. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14(3)** (2003) 5–51
16. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Koranda, S.: Mapping abstract complex workflows onto grid environments. Journal of Grid Computing **1** (2003) 25–39