Modeling Grid Workflows with Colored Petri Nets*

Carmen Bratosin, Wil van der Aalst, and Natalia Sidorova

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
c.c.bratosin@tue.nl, w.m.p.v.d.aalst@tue.nl, n.sidorova@tue.nl

Abstract. Grid computing refers to the deployment of a widely distributed architecture for the execution of computationally challenging tasks. The grid provides a set of distributed resources which can be used for "computing on demand" or for constructing a "virtual supercomputer". Recently, several researchers started to look at the relation between workflow management and grid computing. The flow of work through a grid can be seen as a classical "workflow". However, as opposed to the classical workflows, the resources are not humans and are not managed by some centralized client-server architecture. Instead, the grid is highly distributed and the resources are computing power, memory, etc. Currently, there is no conceptual framework for grid computing and the role of workflows in grids is unclear. This paper provides initial steps towards a conceptual framework expressed in terms of Colored Petri Nets. CPN Tools is used to model grids while focusing on the workflow aspects. The resulting model can be analyzed to detect deadlocks, etc. The framework is illustrated using process mining as an application.

Keywords: Colored Petri nets; grid computing; modeling.

1 Introduction

Grid computing [21] is concerned with the development and advancement of technologies that provide seamless and scalable access to wide-area distributed resources. Currently, many researchers and practitioners are developing software to support grid computing. A well-known example is the *Globus Toolkit* which provides an open source software toolkit for building grids [17]. Within the grid community several research groups have made attempts to adopt ideas from workflow management and apply them in a grid context [15, 18, 19, 23, 25]. For many grid applications the workflow-paradigm is quite natural, e.g., complex scientific computations can be modeled as workflows. However, unlike classical workflows the control is decentralized and resources are computing power, memory, etc. rather than people.

* This research is supported by the GLANCE NWO project "Workflow Management for Large Parallel and Distributed Applications".

Despite the current interest in grids and workflow, *a good conceptual model of grids is missing* and most researchers are focusing on the practical realization of grids. Terms like "resource", "job", and "workflow" are subject to multiple interpretations. Therefore, in this paper, we model the basic grid concepts in terms of *Colored Petri Nets* (CPNs, [20]). The main purpose is to clarify the basic concepts. Moreover, we also show that the mapping of grids onto CPNs allows for all kinds of analysis. Given the fact that the allocation and deallocation of resources in grids is done in a distributed manner and that multiple resources may be involved in some task, a grid workflow may easily deadlock. Therefore, this paper will focus on the use of state-space analysis to discover deadlocks.

Grids are often used in areas where there is a need for a lot of (preferably inexpensive) computing power. Examples can be found in scientific computing, e.g., SETI@home searches for possible evidence of radio transmissions from extraterrestrial intelligence using data from radio telescopes. SETI@home uses CPU-scavenging for this, i.e., a grid of unused desktop computers is exploited to analyze the radio transmissions. This particular form of grid use is also called "voluntary computing" because the resources are made available without a clear economic motive for the participants. Another interesting application domain is the use of grids for data mining to analyse the large volumes of data generated today (cf. the DataMiningGrid project [2]). In this paper, we will focus on a particular application: the utilization of grid computing for *process mining*. The goal of process mining is to extract models (e.g. Petri nets) from event logs [9]. This is possible because many systems ranging from enterprise information systems and web applications to embedded and high-tech systems are collecting enormous volumes of audit trails. To deal with these large amounts of data and computationally expensive process mining algorithms, grids are particularly useful. High-level process mining tasks can easily be described as workflows where the activities correspond to the execution of particular process mining algorithms. Therefore, process mining is an interesting application domain for grid computing.

The remainder of the paper is organized as follows. In Section 2 we present a running example and motivate the utilization of grid computing for process mining. Section 3 introduces the basic grid concepts which are mapped onto CPNs in Section 4. Related work is discussed in Section 6 and Section 7 concludes the paper.

2 Running Example: Applying Grid Technology to Process Mining

As indicated in the introduction, in this paper we focus on using grid technology for large process mining tasks. In recent years, process mining has emerged as a way to analyze systems and their actual use based on the event logs they produce [11]. Note that, unlike classical data mining, the focus of process mining is on concurrent processes and not on static or mainly sequential structures. A

classical example is the α-algorithm [11] which automatically constructs a Petri net based on a set of observed system traces.

Process mining is applicable to a wide range of systems. These systems may be pure information systems (e.g., ERP systems) or systems where the hardware plays a more prominent role (e.g., embedded systems). The only requirement is that the system produces *event logs*, thus recording (parts of) the actual behavior. Information systems such as classical workflow management systems (e.g. Staffware) case handling systems (e.g. FLOWer), PDM systems (e.g. Windchill), middleware (e.g., IBM's WebSphere), hospital information systems (e.g., Chipsoft) record detailed information about the activities that have been executed. Other systems recording events are medical systems (e.g., X-ray machines), production systems (e.g., wafer steppers), copiers, sensor networks, etc. An example is the "CUSTOMerCARE Remote Services Network" of Philips Medical Systems (PMS). This is a worldwide internet-based private network that links PMS equipment to remote service centers. An event that occurs within an X-ray machine (e.g., moving the table, setting the deflector, etc.) is recorded and analyzed. The logging capabilities of the machines of PMS illustrate that event logs are widely available.

The goal of process mining is to extract information (e.g., process models) from these logs, i.e., process mining describes a family of *a-posteriori* analysis techniques exploiting the information recorded in the *event logs*.

Simple algorithms such as the α-algorithm [11] are linear in the size of the event log. However, such algorithms do not perform well on real-life data and their simplicity is misleading for two reasons: (1) more advanced process mining algorithms are needed that require lots of computing power and parameter tuning and (2) the "process of process mining" consists of additional pre- and post-processing steps (filtering, cleaning, merging, conformance checking, etc.). It should be noted that event logs may be huge, e.g., there may be thousands of different cases and there may be thousands of events per case. Logs such as the ones produced by the machines of PMS illustrate the computational challenges. Moreover, some process mining techniques require lots of computing power. Consider for example the genetic process mining algorithms described in [24]. All of the more advanced algorithms have lots of parameters that need to be set. Typically, the algorithms are run with different parameter settings to achieve acceptable results. Hence, different process mining experiments are run iteratively or in parallel. Besides running the core process mining algorithms several pre- and post-processing steps need to conducted.

The main goal of grid computing is to offer wide distributed computing and storage facilities for complex applications. From the observations just made, process mining process can require challenging computational executions, and also has to deal with a large amount of data. Therefore, *process mining is an interesting application domain for grid computing*. On the one hand, there are clear computational challenges that can be addressed through grid computing. On the other hand, the "process of process mining" can be seen as a workflow

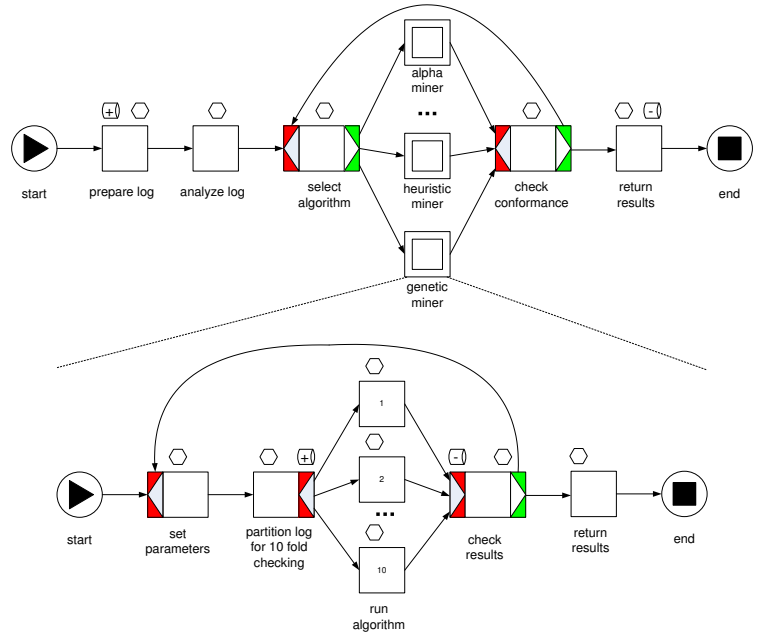

Fig. 1. The process mining workflow expressed in terms of YAWL [6]

consisting of activities ranging from data preparation and filtering to discovery and conformance checking.

To illustrate the application of grid computing to process mining, we use a rather simple and abstract example as shown in Figure 1. It is kept simple so that it is understandable by non-process mining experts and to allow for understandable CPN models later in this paper.

Figure 1 describes a typical process mining scenario in terms of the workflow language YAWL [6]. YAWL extends Petri nets with notations useful for representing workflows. The workflow at the top of Figure 1 shows that a high-level process mining job starts with the preparation of the log (task *prepare log*). It includes the scanning of the log for inconsistencies (e.g., descending timestamps, missing event types, etc.) and the addition of dummy start and end events if needed. Then the log is analyzed (task *analyze log*) and several characteristics are collected, e.g., size, completeness, number of event types, distribution of activities, etc. Based on this task *select algorithm* chooses a particular algorithm that is expected to perform well given the characteristics of the log. If characteristics indicate that the log is highly structured and has no noise, the α-algorithm may be selected. If it contains some noise and is large, the heuristic miner [22] may be selected. The genetic miner may be selected if the structure of the model is complicated, there is noise, and the log is not too large. After running one of the process mining algorithms, the quality of the result is checked (task *check conformance*). If the quality is acceptable or there are no more alternatives,

the results are returned. Otherwise, another algorithm is selected and the process is repeated. Each of the process mining algorithms corresponds to a YAWL subprocess. In Figure 1 only the subprocess *genetic miner* is described. The genetic miner starts by setting the parameters. Genetic algorithms typically have many different parameters that one can experiment with. The genetic miner has parameters such as population size, number of generations, seed, elitism, mutation rate, fitness function, crossover type, etc. Although not shown in Figure 1 different instances of the same algorithm could run in parallel with different parameters to improve response time. After task *set parameters*, the log is split and replicated for 10-fold checking. k-fold cross validation divides the data set into k subsets. Each time, one of the k subsets is used as the test set and the other $k-1$ subsets are put together to form a training set. Then the average error across all k trials is computed. In this workflow the cases in the logs are split over 10 sets and each of the 10 parallel branches in Figure 1 takes 9 of these 10 sets to construct a process model based on the genetic algorithm. After applying the algorithm the result is evaluated using the remaining test set. Task *check results* collects these results and decides whether a new experiment is needed, i.e., the subprocess returns to task *set parameters* or ends with task *return results*.

Figure 1 also shows some annotations describing the use of resources. For this simplified example, we assume that there are only two types of required properties for task execution: CPU and disk space. Disk space is denoted by the small tube and CPU power is denoted by a small hexagon. Disk space is typically allocated for multiple subsequent tasks while CPUs are typically released after each task. Task *prepare log* claims space for storing the entire log and the overall results. This space is only returned at the end of the workflow. Since the genetic algorithm is a more complex process, the algorithm has its own private data space.

In the remainder of this paper, we will use process mining and in particular the example shown in Figure 1 to illustrate our approach.

3 Grid Workflows

This section introduces the basic grid concepts relevant for the remainder of this paper. As indicated, we will model grids in terms of CPNs and emphasize the workflow aspect of grid computing.

The standard grid architecture [16] is composed of several layers: (1) the *infrastructure layer* composed of resources (e.g. databases, cluster computers), (2) the *application layer*, where the grid user describes the processes to be submitted to the grid, and (3) the *middleware layer*, which is in charge of finding a resource for the user requirements and other management issues (e.g. monitoring, fault recovery).

Infrastructure layer The grid infrastructure is a widely distributed infrastructure, composed of different resources, linked via the Internet. The resources allow for the execution of different tasks. Examples of typical resources in a grid infrastructure are *computing elements* (e.g. cluster computers) and *storage elements*

(e.g databases) [3]. A computing element is usually described in terms of its computing power and software available, e.g. number of CPU's, installed software packages, main memory size, operating system. A storage element is a resource that allows grid users to store and manage files together with the space assigned to them. The typical characteristics of a storage element are the software used to manage the device, the allocated space, and, an identifier of the data contained. A storage element typically contains multiple *storage areas*.

We define the resource *capacity* as a set of characteristics that we will refer to as *properties*. Examples are the number of CPU's and HDD size. The capacity of a resource can be described as a multiset of properties, e.g., two CPU's and one disk of 1GB.

Because a resource may host applications according to the available capacity, we split the capacity in *free capacity* (i.e., available computing power or storage to be used by a grid job) and *busy capacity* (i.e., capacity that is already allocated/reserved for the performance of certain jobs). We refer to the set of resources composing a grid infrastructure as the *resource pool*. In the model presented in this paper, we assume that the resource pool is fixed and that the resources are reliable, i.e. if an application was allocated on a resource, then the resource will eventually perform it.

Application layer The upper level of a grid architecture is composed of user applications. Such applications define the jobs to be executed using the grid infrastructure. Since jobs may causally depend on one another, the application level needs to specify the "flow of work". Therefore, we use the term *grid workflow* to refer to the processes specified at the application level. Note that there may be different grid workflows using the same infrastructure and that there may be multiple *instances* of the same grid workflow (referred to as process instances). For each process instance, a partially ordered set of jobs needs to be executed. The grid workflow defines the dependencies between jobs and the properties required per job. In a grid workflow one can find the classical workflow patterns [7] but also patterns focusing on resource allocation, e.g., allocating multiple resources to the same job.

Middleware layer The linking between user jobs and resources is done by a matchmaker (or broker). In this paper we restrict ourselves to middleware working according to a "just-in-time" strategy, i.e., at the moment job instance must be executed, the matchmaker searches for an available resource matching the job, and if it exists the job is allocated to that resource. After the allocation, the free capacity of the resource and the busy capacity are updated according to the job requirements.

In the next section we map the concepts mentioned onto CPNs with two goals in mind: (1) to clarify the basic grid concepts and (2) to show that Petri-net based analysis is useful and feasible in a grid context. Figure 2 illustrates the *grid model* we aim to represent in terms of CPNs. The model is composed of the grid workflows (i.e., application layer) submitted to the grid and a common resource pool, containing all the infrastructure resources with their capacity (i.e.,

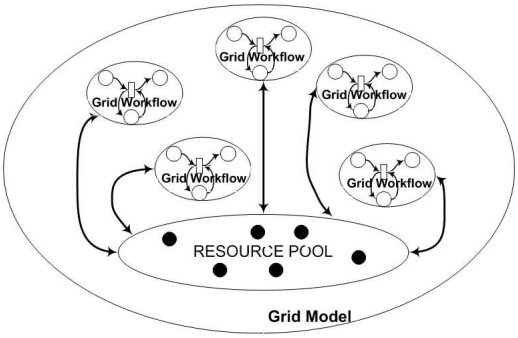

Fig. 2. Grid Model

infrastructure layer). The grid model assumes a very simple middleware layer and will be represented by the allocation/deallocation of the jobs instances only.

4 Modeling Grids in Terms of CPNs

In the previous section, we have presented the main components of a grid model. In this section, we describe how to model a grid using Colored Petri Nets (CPN) [20] and present some basic design patterns [7] that support the modeling of dependencies between grid jobs. We conclude the section by providing a CPN model for the running example presented in Section 2.

4.1 Mapping the grid model onto CPNs

As we discussed in the previous section, a grid model is composed of a set of grid workflows, a pool of resources, and allocation/deallocation mechanisms. In our examples, we typically focus on a single grid workflow, however the same approach can be used to model multiple grid workflows sharing a grid infrastructure.

We model the grid infrastructure in terms of a *resource pool place*. This place contains tokens corresponding to the resources. Each resource has a unique id modeled by the color set *ResID*. The capacity of a resource is expressed in terms of available (*free*) capacity and allocated (*busy*) capacity. Both types of capacity are modeled as a multiset of properties. Recall that a property refers to a single resource characteristic, e.g., a capability like storage space, computing power, bandwidth, etc. The color set *Prop* is used to model properties (e.g. CPU, storage area), and color set *Props* represents a multiset of properties. Note that *Props* is defined as a list of *Prop* elements to model multisets. The tokens of the *resource pool* place are of the color set *Res*. This color set incorporates the resource id, the available capacity, and allocated capacity.

The grid workflows are modeled as an extension of the classical workflow nets [4] and there are also clear relations with the so-called colored workflow

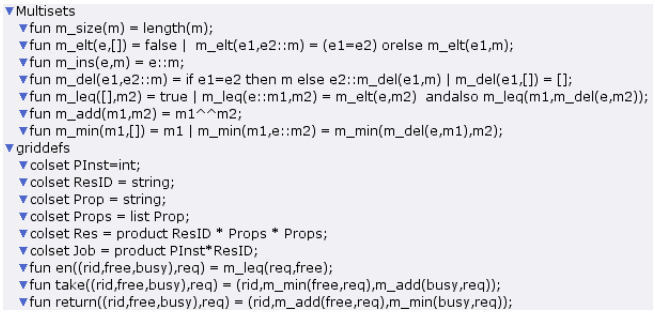

Fig. 3. Color sets for a grid workflow

nets [8]. Like in a workflow net there is a single input *start* place and a single output *end* place, and every node of the grid workflow is on a path from the *start* place to the *end* place. However, we distinguish between two types of places: *job places* and *control places*. Job places correspond to the execution of jobs while using resources from the grid and control places are merely added for the routing of process instances. Job places are mirrored by *requirement* places indicating the resource requirements in terms of a multiset of properties. Another difference with classical workflow nets is that transitions do not represent tasks but correspond to the allocation or deallocation of resources, i.e., we are forced to model the workflow at a finer level of granularity. Initially, all job places and control places are empty and only the requirement places contain tokens. Moreover, for each process instance a token is added to the *start* place.

Process instances are referred to using the color set *PInst*. All control places, including the *start* and *end* place, are of type *PInst*. Job places are of type *Job*. This color set is defined as the product of a process instance id (color set *PInst*) and a resource id (color set *ResID*).[1] Each requirement place contains one token of type *Props*, i.e., the token holds a multiset of properties denoting the resource requirements of the corresponding job place.

In the CPN model we assume a very simple middleware layer, therefore the binding between the grid infrastructure and the grid workflows can be realized through *allocation* and *deallocation* transitions. When a job is created, i.e., a token is put on a job place, an allocation transition fires. If a job completes, i.e., a token is removed from a job place, a deallocation transition fires.

Figure 3 presents all the basic color sets defined for the grid model. In the definitions, we define also the basic operations for multisets. These operations will be used for modeling with allocating and deallocation capacity.

Figure 4 shows a very simple example of a grid model which uses the color sets mentioned before. There is one start and one end place and these are the only two control places. There is just one job place j of type *Job*. The corresponding requirements place is named r and is of type *Props*. The resource

[1] Note that this color set assumes that a job cannot use multiple resources. Later we will relax this requirement.

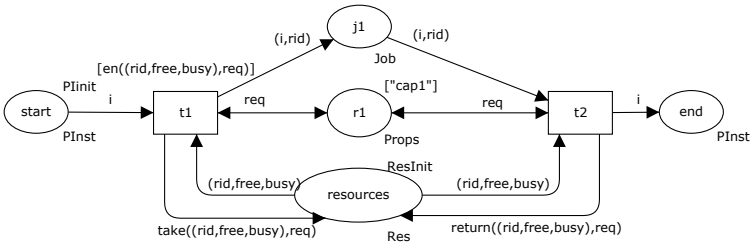

Fig. 4. A simple job example

pool is modeled by the place *resources*. An allocation transition *t1* precedes the job place in Figure 4. The guard of this transition is given by the function *en((rid,free,busy),req)*. The transition is enabled if there exists at least one resource (*rid*) such that the token in *r* place (*req*) is a subset of the multiset *free* (i.e. free resource capacity).

By the firing of transition *t1*, a token containing the process id (i) and the allocated resource id (*rid*) is created for the job place *j*. At the same time, the allocated resource characteristics are retrieved from the resource pool. The token of the resource pool is modified by function *take((rid,free,busy),req)*. The function modifies the capacity occupied by the job as busy capacity.

When the job is finished, the deallocation transition fires (transition *t2* in Figure 4). The function *return((rid,free,busy),req)* modifies the token of the resource that the job releases, by updating the free capacity of the resource (i.e. the new free capacity is the reunion of the free capacity with the capacity equal with the job requirements).

4.2 Basic patterns

In the previous subsection we modeled a grid model containing just one grid workflow and this grid workflow consisted of only one job. However, it is obvious how these types and naming conventions can be used to represent larger grid models. To illustrate this we define some basic patterns to help a grid user to define his process. These are inspired by the workflow patterns in [7]. However, we also provide a pattern dedicated to multiple resource allocation.

Atomic job In the previous section, we have presented a simple job example (see Figure 4). The quadruple *job* place, *requirement* place, *allocation* transition, and *deallocation* transition is the most simple pattern that we use in our model. All the other patterns are composed of this pattern. Therefore, for simplicity we define a subpage containing this pattern.The user can use this subpage in different locations of his grid workflow, adapting the marking of the *req* place according to the job description. Note that for each job type a different subpage can be defined that is reused multiple times.

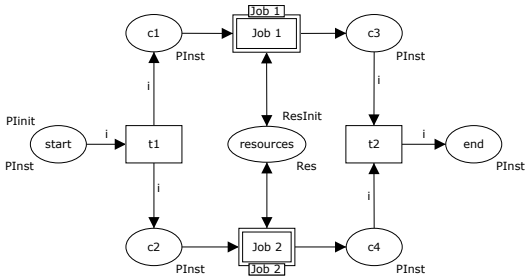

Fig. 5. Parallel pattern

Parallel pattern A common pattern in grid computing is the execution of different jobs in parallel. Figure 5 presents the parallel pattern. Since the matchmaker assumes all the jobs to be independent, two tokens are created so that the matchmaker allocates each job when it finds a suitable resource for this job. The jobs can in principle be different, therefore they are mapped to two different *Job* subpages. The process instance will wait till both jobs finish, and then the transition *t2* fires, which terminates the execution of the pattern.

Multiple resource allocation A typical scenario in grid computing is that multiple resources are needed for the execution of a job. For example, a job requires access to a storage area, that contains some data and, after some computation, the result is written back to the same storage area. The storage area has to be reserved till the computation is finished. Figure 6 illustrates one of the possible patterns to realize this. This pattern creates a job to reserve one of the resources, and then looking for a second resource. The first resource remains locked till the second resource finishes the computation.

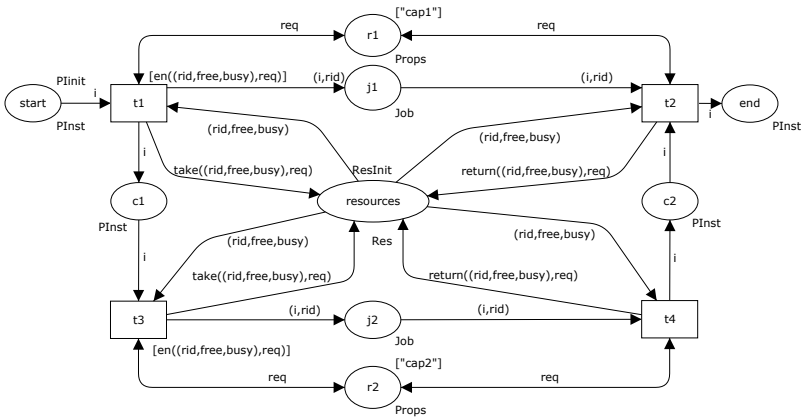

Fig. 6. Multiple resource allocation: Pattern 1

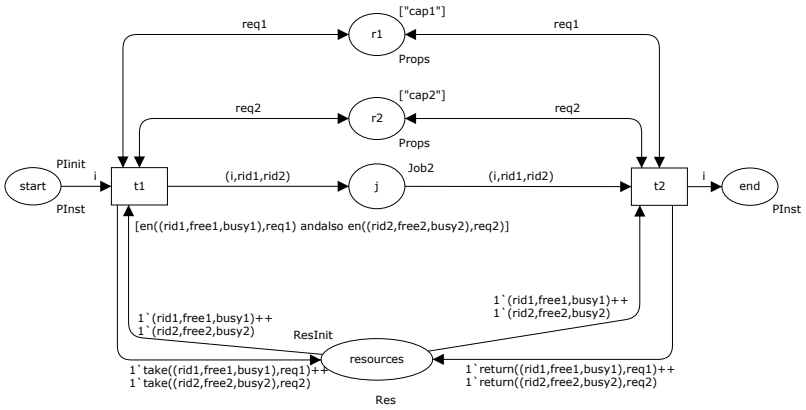

Fig. 7. Multiple resource allocation: Pattern 2

The disadvantage of the pattern illustrated by Figure 6 is that it can lead to deadlocks when a second resource is not available in the resource pool because it is already locked by other jobs.

Figure 7 presents another pattern to realize the allocation of multiple resources. For each of the required resources, the user creates a requirement place: *r1* and *r2*. From the resource pool the allocation transition retrieves two resources, each one corresponding to one of the requirements places. In this case the token of the job place contains one process instance id and two resources ids. The deallocation transition will release both resources. In this second pattern, there may be multiple resources involved in the same job. This makes the model less simple and may lead to modeling errors such as releasing the wrong resource.

4.3 Process mining example mapped to CPN

Figure 8 shows the CPN model of the process mining example already described in Section 2. In Figure 1 this grid model was introduced using a YAWL diagram. Here we focus on the mapping of Figure 1 into Figure 8 using the patterns defined before.

First, a storage area has to be allocated (places *SA allocated* and *req for SA*). The storage area will be used to store and to retrieve the results of all the executed computations. The allocated storage area will be released only when all the other jobs of the same instance have finished. We choose to use the multiple resource pattern shown in Figure 6 to model this behavior.

The sequence of jobs *prepare log* and *analyze log* follows the sequence pattern. After, the *analyze log*, a choice between different mining algorithms should be made. To model this choice we introduce a new color set *ChAlg* that can take three values: *AA*, *HM*, and *GM*. Each of the three values corresponds to the choice of one of the process mining tools: *AA* for the plug-in using the α-

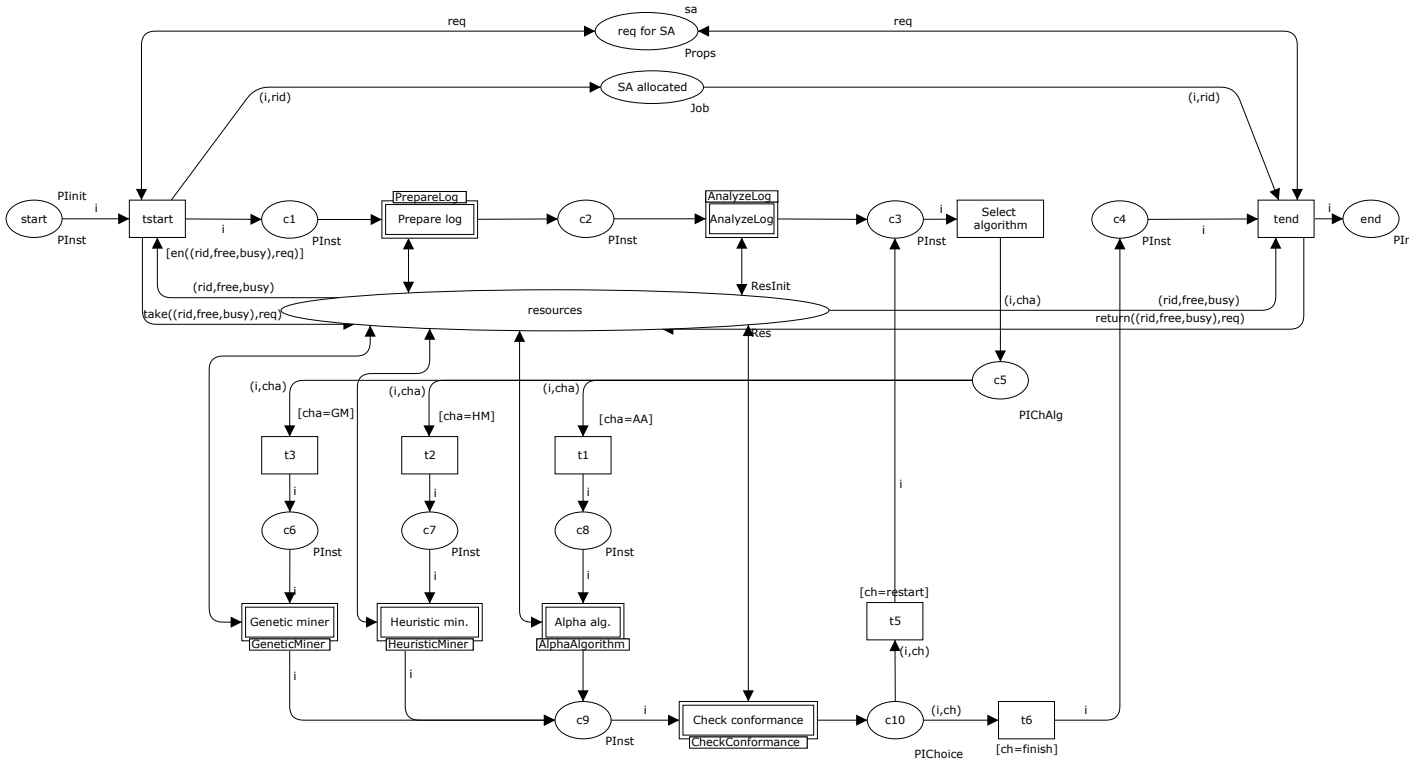

Fig. 8. CPN model of the process mining workflow

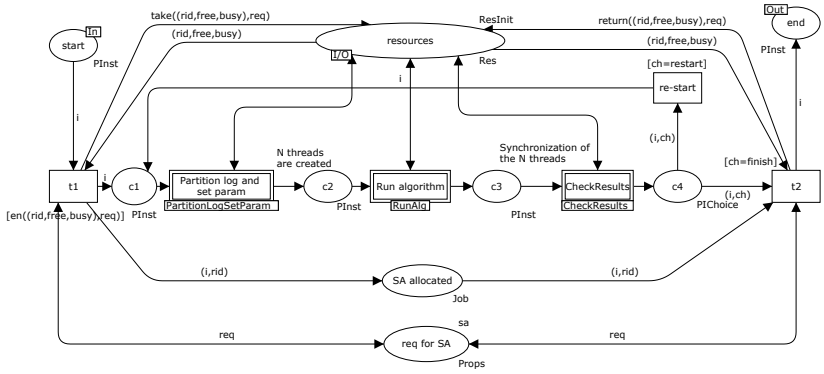

Fig. 9. Genetic miner

algorithm, *HM* for heuristic miner, and *GM* for genetic miner. Note that place *c5* has *PIChAlg* as color set. This color set is the product of *PInst* and *ChAlg*.

After the selected algorithm is executed, job *check conformance* is submitted to the grid. According to the results one of the algorithms (re-)starts or the process ends. To make this choice we use a variable from the color set *Choice*, that can take two values: *restart* to (re-)do one of the algorithms and *finish* to end the process. Note that place *c10* has *PIChoice* as color set. This color set is the product of *PInst* and *Choice*.

For the execution of the α-algorithm and heuristic miner, a resource with free *CPU* capacity is needed. On the other hand, the use of genetic algorithms requires a more complex process (cf. Figure 9). A new storage area is allocated in order to store/retrieve intermediate results of the genetic miner. Then a computing element is used to partition the log and to set the parameters. N different threads are created inside subpage *PartitionLogSetParam* (not shown here), and on each thread the genetic miner algorithm is performed (a grid job requiring a computing element). When all threads have finished, the results of the threads are combined and analyzed (job *check results*). After the job *check results* ends, all the process re-runs or it stops (for this choice an element of color set *Choice* is used).

The example CPN model shows that it is indeed possible to model complex computations using a grid infrastructure. Using CPN Tools we were able to clarify the basic terms used in grid computing. Moreover, the allocation and deallocation of resources may lead to deadlock problems as will be shown in the next section. Because of this, the CPN model is also useful from an analysis point of view.

5 Analysis of Grid Workflows

The analysis of grid workflows may include *validation* (i.e. checks whether the workflow behaves as expected), *verification* (i.e. workflow correctness) and *per-*

formance analysis (i.e. evaluating whether time or cost requirements are fulfilled). In this section we focus on a specific type of analysis: verification. The verification of grid workflow is related to the correctness properties such as absence of deadlocks, livelocks and resource conflicts (cf. [14]). To verify these properties, we use the state space analysis functionality provided by CPN Tools. The analysis is conducted in two steps. First, we perform a *soundness* check. For this purpose, we will extend the soundness property [10] for traditional workflow such that it takes into account the grid workflow characteristics. In the second step, we verify whether there are any resource conflicts between different jobs originating from different grid workflows and their instances. We also show a more efficient deadlock analysis technique which allows us to look at one instance in isolation.

5.1 Soundness check

Since we are interested in soundness, we assume in this subsection that the resource pool has an "infinite" amount of resources (i.e. whenever a job claims resources, available resources exist in the resource pool).

The correctness property that we want to establish is *soundness*. A traditional workflow is sound if it satisfies the following property: if we put a token in the *start* place, there is a possible path to reach the *end* place with just one token from each reachable state, and if the *end* place is reached no "garbage" tokens are left behind (i.e. all the places except the *end* place are unmarked). In our case, because the grid workflow contains also *requirements* places and a *resource* place, we have to extend the soundness property with the following two conditions: (1) the marking of the requirements places remains the same for all the reachable markings and (2) the resource pool place marking in the final state has the same value as in the initial state.

The necessary and sufficient conditions that ensure that a grid workflow is sound using the state space report of CPN Tools are the following: (1) there exists only one dead marking, this marking should also be a home marking (i.e. a marking reachable from all other markings) and there are no other home markings, moreover this dead marking is indeed the desired final marking (i.e. the marking meets the soundness conditions) and (2) the lower and upper bounds of a requirement place marking are equal.

For Figure 8, CPN Tools generated a full state space of 91 nodes and 139 arcs in less than one second for an initial state with just one process instance and a resource pool tokens[2]: `1‘("CE",["CPU","CPU","CPU","CPU","CPU","CPU","CPU","CPU","CPU","CPU"],[])++ 1‘("SE",["SA","SA","SA"],[])`.

The state space report provided information about boundedness properties, home properties and liveness properties as below[3]:

[2] Note that we did not add infinitely many resources to the initial marking. However, it is easy to check whether sufficient resources have been added by checking the multi set bounds in the state space report.

[3] We present a partial state space report that contains only the necessary information to establish grid workflow soundness.

```
Boundedness Properties
----------------------
 Best Integer Bounds
                         Upper  Lower
  AlphaAlgorithm'r 1       1      1
  AnalyzeLog'r 1           1      1
  CheckConformance'r 1     1      1
  CheckResults'r 1         1      1
  GeneticMiner'req_for_SA 1       1
  HeuristicMiner'r 1       1      1
  PM'req_for_SA 1          1      1
  PartitionLogSetParam'r   1      1
  PrepareLog'r 1           1      1
  RunAlg'r 1               1      1
 Best Upper Multi-set Bounds
  AlphaAlgorithm'r 1  1`["CPU"]
  AnalyzeLog'r 1      1`["CPU"]
  CheckConformance'r 1
    1`["CPU"]
  CheckResults'r 1    1`["CPU"]
  GeneticMiner'req_for_SA 1
    1`["SA"]
  HeuristicMiner'r 1  1`["CPU"]
  PM'req_for_SA 1     1`["SA"]
  PartitionLogSetParam'r 1
    1`["CPU"]
  PrepareLog'r 1      1`["CPU"]
  RunAlg'r 1          1`["CPU"]
 Best Lower Multi-set Bounds
  AlphaAlgorithm'r 1  1`["CPU"]
  AnalyzeLog'r 1      1`["CPU"]
  CheckConformance'r 1
    1`["CPU"]
  CheckResults'r 1    1`["CPU"]
  GeneticMiner'req_for_SA 1
    1`["SA"]
  HeuristicMiner'r 1  1`["CPU"]
  PM'req_for_SA 1     1`["SA"]
  PartitionLogSetParam'r 1
    1`["CPU"]
  PrepareLog'r 1      1`["CPU"]
  RunAlg'r 1          1`["CPU"]

Home Properties
----------------------
 Home Markings
   [28]

Liveness Properties
----------------------
 Dead Markings
   [28]
 Dead Transition Instances
   None
 Live Transition Instances
   None
```

From the boundedness properties, we observe that each of the requirement places (i.e. r places) has a lower bound equal to the upper bound. Marking 28 is a home marking and a dead marking and it corresponds to the desired final marking. Therefore, we conclude that the process mining grid workflow is sound.

5.2 Resource verification

The main problem in resource sharing is the potential of deadlocks when multiple instances run in parallel and compete for the same resources step-by-step. Therefore, we now look at the analysis of a grid model with *multiple instances* and a *limited set of resources*. Using CPNTools we want to verify whether soundness is jeopardized. We do this by looking for deadlocks.

Let us consider the process mining grid workflow again. For two process instances and a resource pool containing the following tokens:

```
1`("CE1",["CPU","CPU","CPU"],[])++1`("SE1",["SA"],[])++1`("SE2",["SA"],[])
```

CPN Tools generates a full state space with 1140 nodes and 2176 arcs in less than 2 seconds. The state space report contains three dead markings and no home markings.

```
 Home Properties                    Liveness Properties
-----------------------            -----------------------

  Home Markings                      Dead Markings
     None                               [420,456,952]
```

Dead marking 952 is the desired final marking (i.e. the grid model proper terminates), and the other two (420,456) refer to instances where both of process instances deploy the genetic miner concurrently. The deadlocks occur because each of the instances needs an additional storage area, but the resource pool depleted all available storage areas. Even if we increase the number of available resources, the system will deadlock when the number of process instances running in parallel is also increased.

In the following subsection, we propose a method to correct such a grid workflow in order to ensure its proper termination independently from the number/type of available resources.

5.3 Correcting a resource constraint grid workflow

In [26], we studied resource-constraint processes with homogeneous resources and we have shown that a necessary and sufficient condition that ensures proper termination (i.e. absence of deadlocks violating e.g. the soundness property) is that any path resulting in the claim of one or more resources should have a successor path resulting in the release of some resources.

We verify the necessary and sufficient condition for each property type. Therefore, we construct an automaton modeling the behavior of the system just from the point of view of claiming and releasing of resources, abstracting from all the the other possible events (i.e. those events are considered as silent steps). Figure 10 presents the automaton for the process mining workflow. In state *s1*, the system needs to reserve a computing element with a free *CPU* capacity, and in state *s3* a new storage area if the genetic miner algorithm is selected. The two claims are made without being preceded by releases of resources providing the same type of property. Therefore, the workflow does not satisfy the necessary

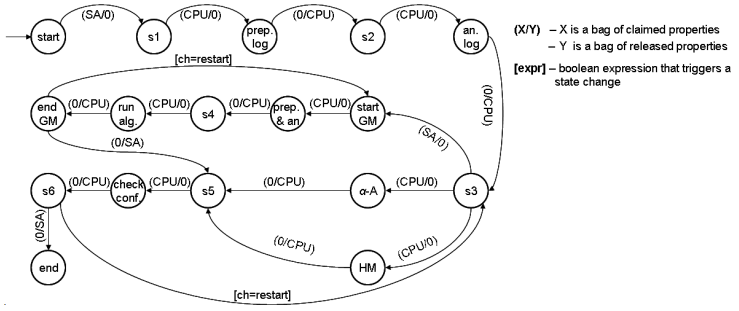

Fig. 10. Process mining automaton

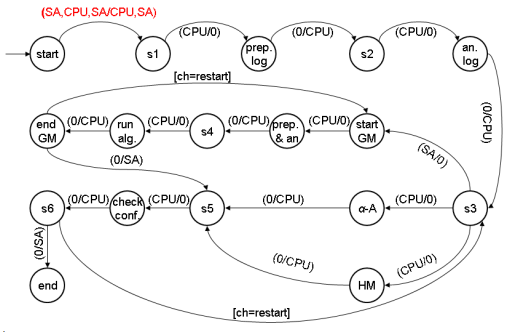

Fig. 11. Modified process mining automaton

and sufficient condition (presented in [26]) neither for *CPU* property, neither for *SA* property.

To avoid deadlocks the model shown in Figure 8 needs to be corrected. The correction of the grid workflow is made by checking if there are enough resources to execute all the jobs composing the workflow (cf. [26]). The resources are just claimed and released, ensuring that the necessary and sufficient condition is satisfied. The algorithm presented in [26] is applied for each property type.

For the running example, the corrected automaton, by joining the results for each property type, is shown in Figure 11. The only modification made is to claim and to release in state *start* two additional resources with properties *SA* and *CPU*. For the corrected automaton, we observe that the necessary and sufficient condition is fulfilled. We map the solution from the automaton to the

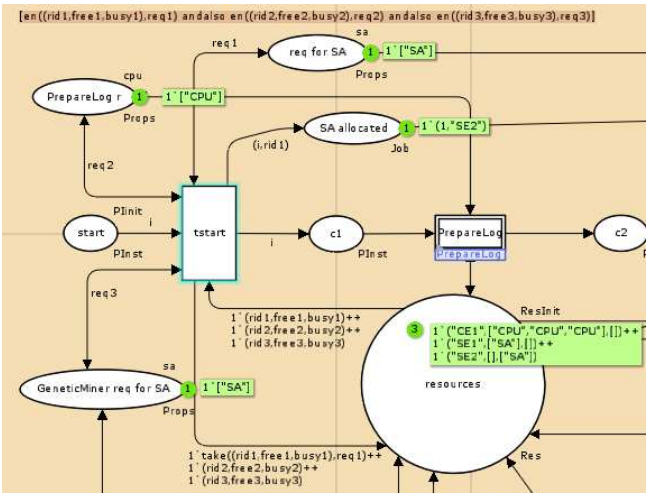

Fig. 12. The changed process mining workflow

CPN model, by changing the guard of the allocating transition of the first storage area as shown in Figure 12. From the *resource pool*, three resources are required (*rid1*, *rid2* and *rid3*). Each of this resources must fulfill one of the *requirements* of the jobs contained by the workflow. Just one resource is kept (*rid1*, as in the original model), and the other two are released without any modification.

For the modified model, CPN Tools generates a full state space with 492 nodes and 664 arcs in less than one second. Just one marking is reported as both a home marking and dead marking and it corresponds to the desired final marking. Hence the grid model is sound.

6 Related Work

The idea of using grid workflows to model and enact complex scientific processes and distributed resources such as computing elements and data has emerged in the grid community in recent years. Grid workflow systems [1, 15, 25] have been developed, but most of them support only directed acyclic graphs (DAGs) as a modeling language. The disadvantage of using a DAG is that it does not support loop patterns and choice patterns. Therefore, in the last years, Petri Nets were introduced [12, 19] as a more powerful and suitable language to model grid workflows. However, most of the work is focused on the practical realization of grids, and less attention has been paid to providing a conceptual framework to model grid workflows.

Many reseachers have applied Petri nets to workflow management [4, 27]. Note that these papers do not necessarily focus on grid workflows. Moreover, these papers typically focus on a single aspect, e.g. control flow verification, while ignoring the interplay between resources and workflows. However, there have been some papers using CPNs to address other aspects of workflows, e.g. [8]. To address the deadlock problem in grids we propose to use a variant of the technique proposed in [26].

The running example comes from the process mining domain. In [13], the authors proposed a process mining workflow that can call process mining algorithms using a process engine. More details related to process mining concepts and algorithms can be found in [11, 22, 24].

7 Conclusion

In this paper, we propose a conceptual framework to use CPN for modeling and verifying of grid workflows. Grid concepts such as "job", "grid workflow" and "resource" have been explained in order to map them on CPNs.

We proposed some basic patterns in order to model common grid behavior such as parallel execution and multiple resources allocation. Combining these patterns, a complex grid workflow was build up.

Moreover, we showed that CPN Tools can be used to conduct soundness and resource verification to find potential deadlocks. The state space analysis from

CPN Tools is able to discover deadlocks when multiple instances run in parallel while incrementally claiming similar resources. Based on our previous work [26], we have corrected the model such that all instances terminate properly.

This paper is mainly conceptual. However, it is good to mention that we are in the process of connecting YAWL, Globus, and ProM. YAWL [6] is used as a workflow engine taking care of the orchestration. Globus [17] is an open source software toolkit used for building Grid systems and applications but is not process-aware. ProM [5] is used to execute the actual process mining activities. ProM provides a plugable architecture with dozens of process mining algorithms and a lot of functionalities related to filtering, conversion, conformance checking, etc. Currently, ProM [5] aims at interactive mining. However, as shown in [13], ProM can be adapted such that its functionality can be invoked by a workflow engine.

References

1. DAGMan (Directed Acyclic Graph manager): Condor meta-scheduler. http://www.cs.wisc.edu/condor/dagman/.
2. DataMiningGrid Project. http://www.datamininggrid.org/.
3. GLUE Schema V1.3. http://forge.gridforum.org/sf/go/doc14185?nav=1.
4. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
5. W.M.P. van der Aalst, B.F. van Dongen, C. Günther, R. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H. Verbeek, and A. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *ICATPN 2007*, volume 4546 of *LNCS*, pages 484–494. Springer-Verlag, Berlin, 2007.
6. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
7. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
8. W.M.P. van der Aalst, J. Jørgensen, and K. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System Paper. In R. Meersman and Z. T. et al., editors, *OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *LNCS*, pages 22–39. Springer-Verlag, Berlin, 2005.
9. W.M.P. van der Aalst, B. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
10. W.M.P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
11. W.M.P. van der Aalst, A. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
12. M. Alt, S. Gorlatch, A. Hoheisel, and H.-W. Pohl. A Grid Workflow Language using High-Level Petri Nets. Technical Report TR-0032, Institute on Grid Information and Monitoring Services, CoreGRID - Network of Excellence, March 2006.

13. L. Cabac and N. Knaak. Process mining in Petri net-based agent-oriented software development. In D. Moldt, F. Kordon, K. van Hee, J.-M. Colom, and R. Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 7–21, Siedlce, Poland, June 2007. Akademia Podlaska.
14. J. Chen and Y. Yang. Key research issues in grid workflow verification and validation. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 97–104, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.
15. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the Grid. *Grid Computing: LNCS*, 3165:11–20, 2004.
16. I. Foster. The anatomy of the grid: Enabling scalable virtual organizations. *Proceedings. First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 6–7, 2001.
17. I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin, D. A. Reed, and W. Jiang, editors, *NPC*, volume 3779 of *LNCS*, pages 2–13. Springer, 2005.
18. G. C. Fox and D. Gannon. Workflow in Grid Systems. *Concurrency and Computation: Practice and Experience*, 18(10):1009–1019, 2006.
19. A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency and Computation: Practice and Experience*, 18(10):1101–1113, 2006.
20. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
21. C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.
22. L. Maruster, A. Weijters, W.M.P. van der Aalst, and A. van den Bosch. A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs. *Data Mining and Knowledge Discovery*, 13(1):67–87, 2006.
23. A. S. McGough, W. Lee, and J. Darlington. Workflow deployment in ICENI II. In *International Conference on Computational Science (3)*, pages 964–971, 2006.
24. A. Medeiros, A. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: A Basic Approach and its Challenges. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *LNCS*, pages 203–215. Springer-Verlag, Berlin, 2006.
25. T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
26. K. M. van Hee, A. Serebrenik, N. Sidorova, M. Voorhoeve, and J. van der Wal. Scheduling-free resource management. *Data and Knowledge Engineering*, 61(1):59–75, 2007.
27. Y. Wang, C. Lin, Y. Yang, and Y. Qu. Grid service workflow models and their equivalent simplification methods. In *GCC Workshops*, pages 302–307. IEEE Computer Society, 2006.