Performance Metrics and Ontology for Describing Performance Data of Grid Workflows *

Hong-Linh Truong
Institute for Software Science, University of Vienna
truong@par.univie.ac.at

Thomas Fahringer, Francesco Nerieri
Institute for Computer Science, University of Innsbruck
{tf,nero} @dps.uibk.ac.at

Schahram Dustdar
Information Systems Institute, Vienna University of Technology
dustdar@infosys.tuwien.ac.at

Abstract

To understand the performance of Grid workflows, performance analysis tools have to select, measure and analyze various performance metrics of the workflows. However, there is a lack of a comprehensive study of performance metrics which can be used to evaluate the performance of a workflow executed in the Grid. This paper presents performance metrics that performance monitoring and analysis tools should provide during the evaluation of the performance of Grid workflows. Performance metrics are associated with many levels of abstraction. We introduce an ontology for describing performance data of Grid workflows. We describe how the ontology can be utilized for monitoring and analyzing the performance of Grid workflows.

1. Introduction

Recently, increased interest can be witnessed in exploiting the potential of the Grid for workflows, especially for scientific workflows, e.g. [15, 2, 8]. As the Grid is diverse, dynamic and inter-organizational, the execution of Grid workflows is very flexible. This requires performance monitoring and analysis tools to collect, measure and analyze metrics that characterize the performance and dependability of workflows at many levels of detail in order to detect components that contribute to performance problems, and correlations between them.

To understand the performance and dependability of Grid workflows, performance metrics of the workflows have to be studied and defined. However, there is a lack of a comprehensive study of useful performance metrics which can be used to evaluate the performance of workflows executed in the Grid. Only a few metrics are supported in most existing tools and most of metrics are being limited to the activity (task) level. Moreover, performance data of workflows not only is used for reasoning performance problems but also needs to be shared because various other tools, such as workflow composition tools, schedulers and optimization tools, require the performance data. Therefore, an ontology describing performance data of workflows is important because the ontology will facilitate the performance data sharing and can be used to explicitly describe concepts associated with workflow executions.

Previously, we have developed an ontology to describe performance data of Grid applications [18]. This paper extends our previous work to study performance metrics of Grid workflows and to describe performance data of the Grid workflows. We propose an extended set of performance metrics associated with multiple levels of abstraction; these metrics characterize the performance of Grid workflows. Proposed performance metrics are described in a metric ontology. We then introduce an ontology which can be used to describe performance data of Grid workflows. The ontology is intended to establish a common understanding about the performance of Grid workflows thus it can be shared and used by various tools and services.

The rest of this paper is organized as follows: Section 2 discusses the workflow and workflow execution model. Section 3 presents performance metrics for workflows. We introduce an ontology for describing performance data of workflows in Section 4. We discuss the use of the ontology for performance analysis of Grid workflows in Section 5. Related work is outlined in Section 6. We summarize the paper and give an outlook to the future work in Section 7.

*The work described in this paper is supported in part by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104 and by the European Union through the IST-2002-511385 project K-WfGrid.

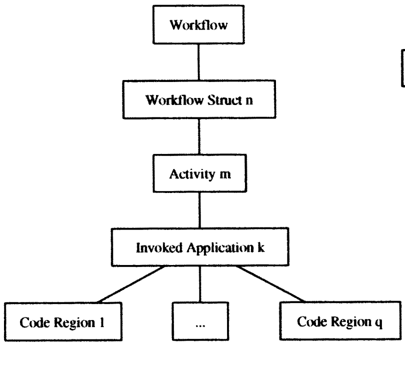

Figure 1. Hierarchical structure view of a workflow.

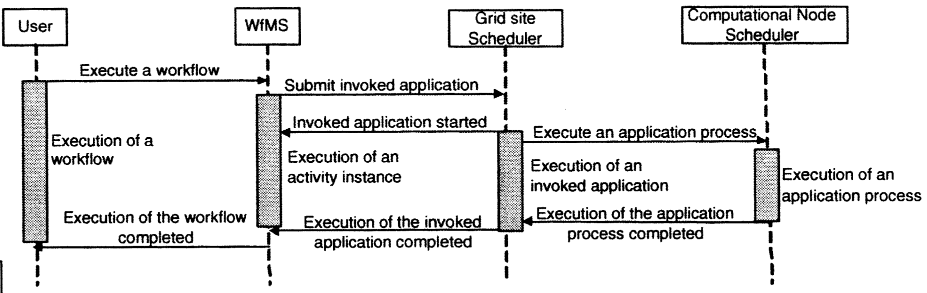
User
WfMS
Grid site Scheduler
Computational Node Scheduler
Execute a workflow
Submit invoked application
Invoked application started
Execute an application process
Execution of a workflow
Execution of an activity instance
Execution of an invoked application
Execution of an application process
Execution of the workflow completed
Execution of the invoked application completed
Execution of the application process completed

Figure 2. Execution model of a workflow.

2. Workflow Model

2.1. Hierarchical Structure View of a Workflow

Figure 1 presents the hierarchical view of a workflow (WF). A WF consists of WF constructs. Each WF construct consists of a set of activities. Two activities can depend on each other. The dependency between two activities can be *data dependency* or *control dependency*. Each activity is associated with a set of invoked applications. Each invoked application contains a set of code regions.

WF constructs can be fork-join, sequence, do loop, etc. More details of existing WF constructs can be found in [1]. Each activity is associated with one or multiple invoked application(s). An invoked application can be an executable program (e.g., an MPI program) or a service operation (e.g., of Web Service). Invoked applications can be executed in sequential or parallel manner. An invoked application is considered as a set of code regions; a code region ranges from a single statement to an entire program unit. A code region can be a function call, a remote service call, a do loop, an if-then-else statement, etc.

2.2. Workflow Execution

A Grid environment is viewed as a set of Grid sites. A *Grid site* is comprised of a set of grid services within a single organization. A Grid site consists of a number of *computational nodes* (or hosts) that share a common security domain and exchange data through a local network. A computational node can be any computing platform, from a single-processor workstation to an SMP (Symmetric Multi-Processor) node to an MPP (Massively Parallel Processing) system. Each computational node may have single or multiple *processor(s)*. On each computational node, there would be multiple *application processes* executed, each process may have multiple *threads* of execution.

Figure 2 presents the execution sequence of a WF. The user submits a WF to the workflow management system (WfMS). The WfMS instantiates activities. When executing an activity instance, the WfMS locates a Grid site and submits the invoked application of the activity instance to the scheduler of the Grid site. The Grid site scheduler locates computational nodes and executes processes of the invoked application on corresponding nodes.

2.3. Activities Execution Model

The execution of an activity a is represented by the discrete process model [14]. Let $P(a)$ be a discrete process modeling the execution of activity a (hence, we call $P(a)$ the *execution status graph* of an activity). A $P(a)$ is a directed, acyclic, bipartite graph (S,E,A), in which S is a set of nodes representing *activity states*, E is a set of nodes representing *activity events*, and A is a set of edges representing ordered pairs of activity state and event. Simply put, an agent (e.g. workflow invocation and control) causes an event (e.g. execute an activity) that changes the activity state (e.g. from queuing to processing), which in turn influences the occurrence and outcome of the future events (e.g. active, failed). Figure 3 presents an example of a discrete process modeling the execution of an activity.

Event Name	Description
active	indicate the activity instance has been started to process its work.
completed	indicate the execution of the activity instance has completed.
suspended	indicate the execution of the activity instance is suspended.
failed	indicate the execution of the activity instance has been stopped before its normal completion.
submitted	indicate the activity has been submitted to the scheduling system.

Table 1. Example of event names.

Each state s of an activity a is determined by two events: leading event e_i, and ending event e_j such that $e_i,e_j \in E$, $s \in S$, and $(e_i,s),(s,e_j) \in A$ of $P(a)$. To denote an event

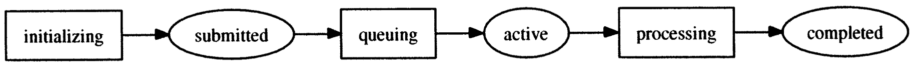

Figure 3. Discrete process model for the execution of an activity. ▭ represents a state, ○ represents an event.

Category	Metric Name	Description
Execution time	ElapsedTime	The elapsed time of the code region.
	UserCPUTime	CPU time spent on user mode
	SystemCPUTime	CPU time spent on system mode
	CPUTime	CPU consumption time. This metric includes CPU times spent on user and system mode.
	SerialTime	The time spent on serialization and deserialization data.
	EncodingTime	The time spent on encoding and decoding data.
Counter	L2_TCM, L2_TCA, etc.	Hardware counters. The exact number of hardware counters is dependent on specific platforms.
	NCalls	The number of executions of the code region.
	NSubs	The number of executions of sub regions of the code region.
	SendMsgCount	The number of messages sent by the code region.
	RecvMsgCount	The number of messages received by the code region.
Data movement	TotalCommTime	Communication time.
	TotalTransSize	Size of total data transfered (send and receive).
Synchronization	ExclSynTime	Single-address space exclusive synchronization.
	CondSynTime	Condition synchronization.
Ratio	MeanElapsedTime	Mean elapsed time per execution of the code region.
	CommPerComp	Communication per computation.
	MeanTransRate	Mean of transfer rate.
	MeanTransSize	Transfered data size per number of transfers.
	CacheMissRatio,MLOPS, etc.	Ratio metrics computed based on hardware counters.
Temporal overhead	octrp,olopa, etc.	This type of metrics is defined only for code regions of parallel programs.

Table 2. Performance metrics at code region level.

name of $P(a)$ we use $e_{name}(a)$; Table 1 presents a few event names which can be used to describe activity events. We use $t(e)$ to refer to the timestamp of an event e and t_{now} to denote the timestamp at which the analysis is conducted. Because the monitoring and analysis of Grid workflows is normally conducted at runtime, it is possible that an activity a is on a state s but there is no such $(s,e) \in A$ of $P(a)$. When analyzing such state s, we use t_{now} as a timestamp to determine the time spent on state s. The *happened before* relation between events is denoted by $\rightarrow$.

3. Performance Metrics of Grid Workflows

The task of performance monitoring and analysis of Grid WFs is to collect and analyze performance metrics related to the WFs. Interesting performance metrics of WFs are associated with many levels of abstraction. We classify performance metrics according to five levels of abstraction, from lower to higher level, including *code region, invoked application, activity, workflow construct* and *workflow*.

In principle, from performance metrics of a lower-level, similar metrics can be constructed for the immediate higher-level by using various aggregate operators such as sum and average. For example, the communication spent in one application may be defined as the sum of communication spent on its code regions. Exact aggregate methods are dependent on specific metrics and their associated levels. In the following sections we present performance metrics with their associated levels. For a higher-level, *we will not show metrics that can be aggregated from that of the lower-level.* Instead, we discuss new metrics which appear at the higher level or an existing metric but it requires a different computing method at different levels of abstraction.

3.1. Metrics at Code Region Level

Table 2 presents performance metrics of code regions. Performance metrics are categorized into: *execution time, counter, data movement, synchronization, ratio* and *temporal overhead.*

Execution time metrics include total elapsed time (wall-clock time or response time)[1], user CPU time, system CPU time, etc. Counter metrics include hardware counters (e.g. L2 cache misses, the number of floating point instructions) and other counters such as the number of calls. Data movement metrics characterize the data movement such as communication time and exchanged message size. Synchronization metrics describe time spent on the synchronization of executions, such as critical section, condition synchronization, etc. Various ratio metrics can be defined based on execution time and counter metrics.

If the invoked application is a parallel application (e.g., MPI and OpenMP applications), we can compute *temporal overhead* metrics for code regions. Overhead metrics are based on a classification of temporal overheads for parallel programs [17]. Examples of overhead metrics are control of parallelism, loss of parallelism, etc.

3.2. Metrics at Invoked Application Level

Most performance metrics at code region level can be provided at invoked application level by using aggregate operators. Table 3 presents extra performance metrics as-

[1]Elapsed time, wall-clock time, and response time indicate the latency to complete a task (including IO, waiting time, computation, ...). These terms are used interchangeably. In this paper, the term *ElapsedTime* refers to elapsed time or response time or wall-clock time.

sociated with invoked applications.

Category	Metric Name	Description
Execution time	ElapsedTime	The elapsed time of the invoked application.
	ExecDelay	The delay between when the Grid scheduler receives a request for creating a new instance of the invoked application and when the instance is created.
Counter	NCalls	The number of executions of the invoked application.
Performance improvement	SpeedupFactor	Speedup factor between executions of the same application.

Table 3. Performance metrics at invoked application level.

Let A be an invoked application. Let $ElapsedTime_i(A)$ and $ElapsedTime_j(A)$ be elapsed times of A in executions i and j, respectively. The speedup factor of execution i over execution j is defined by

$$SpeedupFactor = \frac{ElapsedTime_i(A)}{ElapsedTime_j(A)} \quad (1)$$

3.3. Metrics at Activity Level

Table 4 presents metrics measured at activity level. Performance metrics can be associated with activities and activity instances.

Execution time metrics includes end to end response time, processing time, queuing time, suspending time, etc. The processing time of an activity instance a, $ProcessingTime(a)$, is defined by

$$ProcessingTime(a) = t(e_{completed}(a)) - t(e_{active}(a)) \quad (2)$$

if $e_{completed}(a)$ has not occurred, it means the execution of a has not completed, processing time is defined by

$$ProcessingTime(a) = t_{now} - t(e_{active}(a)) \quad (3)$$

Synchronization metrics for an activity involves with the execution of other activities it depends. Let $pred(a)$ be the set of the immediate predecessors of a; there is a data dependency or control dependency between a and any $a_i \in pred(a)$. $\forall a_i \in pred(a); i = 1, \cdots, n$; synchronization delay and execution delay from a_i to a, $SynDelay(a_i, a)$ and $ExecDelay(a_i, a)$, respectively, are defined by:

$$SynDelay(a_i, a) = t(e_{submitted}(a)) - t(e_{completed}(a_i)) \quad (4)$$

$$ExecDelay(a_i, a) = t(e_{active}(a)) - t(e_{completed}(a_i)) \quad (5)$$

If $e_{submitted}(a)$ or $e_{active}(a)$ has not occurred, synchronization or execution delay will be computed based on t_{now}.

Metrics associated with an activity are determined from metrics of activity instances of the activity by using aggregate operators. Aggregated metrics of an activity give the summarized information about the performance of the activity that can be used to examine the overall performance of the activity.

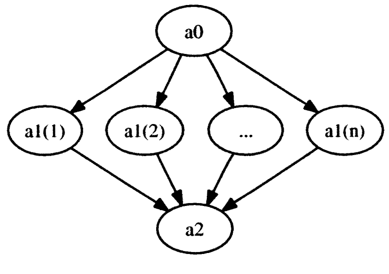

Figure 4. A fork-join workflow construct.

3.4. Metrics at Workflow Construct Level

Table 5 presents performance metrics at WF construct level. The load imbalance is associated with fork-join WF constructs. A fork-join WF construct is shown in Figure 4. Load imbalance, $LoadIm$, is defined by

$$LoadIm(a_i) = ProcessingTime(a_i) - \frac{\sum_{k=1}^{n}(ProcessingTime(a_i))}{n} \quad (6)$$

Speedup factor for a fork-join construct is defined by

$$SpeedupFactor = \frac{ProcessingTime_1(a_i)}{max_{i=1}^{n}(ProcessingTime_n(a_i))} \quad (7)$$

where $ProcessingTime_n(a_i)$ is the processing time of activity a_i in the fork-join version with n activities and $ProcessingTime_1(a_i)$ is the processing time of activity a_i in the version with a single activity. Load imbalance and speedup factor metrics can also be computed for fork-join structures of *structured block* of activities. A structured block is a single-entry-single-exit block of activities. In this case, $ProcessingTime_n(a_i)$ will be the processing time of a structured block in a version with n blocks.

Let SG be a graph of WF construct C. Let $P_i =< a_{i1}, a_{i2}, \cdots, a_{in} >$ be a critical path from starting node to the ending node of of SG. The elapsed time of C, $ElapsedTime(C)$, and the processing time of C, $ProcessingTime(C)$, are defined as

$$ElapsedTime(C) = \sum_{k=1}^{n} ElapsedTime(a_{ik}) \quad (8)$$

$$ProcessingTime(C) = \sum_{k=1}^{n} ProcessingTime(a_{ik}) \quad (9)$$

Now, let C_g and C_h be WF constructs of a workflow-based application; C_g and C_h may be identical construct but be executed on different resources at different times. Speedup factor of C_g over C_h, $SpeedupFactor(C_g, C_h)$, is defined by

$$SpeedupFactor(C_g, C_h) = \frac{ProcessingTime(C_g)}{ProcessingTime(C_h)} \quad (10)$$

3.5. Metrics at Workflow Level

Table 6 presents performance metrics of interest at WF level. Let $P_i =< a_{i1}, a_{i2}, \cdots, a_{in} >$ be a critical path from starting node to the ending node of a WF G. The elapsed

Category	Metric Name	Description
Execution time	ElapsedTime	End-to-end response time of an activity instance.
	ProcessingTime	The time an activity spends on the processing. Processing time includes both communication and computation times.
	QueuingTime	The time an activity is on queuing system.
	SuspendingTime	The time an activity spends on suspended state.
	SharedResTime	The period of time on which the activity has to share the resource with other activities.
Counter	NCalls	The number of invocations of an activity.
	InTransSize	Size of total data transfered to the activity per data dependency.
	OutTransSize	Size of total data transfered from the activity to another.
Ratio	Throughput	The number of successful activity instances over time.
	MeanTimePerState	The mean time that an activity spent on a state.
	TransRate	Data transfer rate per data dependency.
Synchronization	SynDelay	Synchronization delay.
	ExecDelay	Execution delay.
Performance improvement	SlowdownFactor	Slowdown factor is the ratio of ElapsedTime to ProcessingTime.

Table 4. Performance metrics at activity level.

Category	Metric Name	Description
Execution time	ElapsedTime	The latency from the time the workflow construct starts until the time the workflow construct finishes.
	ProcessingTime	The actually portion of elapsed time that the workflow construct spends on processing.
Counter	RedundantActivity	The number of activity instances whose processing results are not utilized (e.g. in a *discriminator* construct).
	NIteration	The number of iterations of a loop construct.
Ratio	MeanElapsedTime	The average elapsed time per activity of the workflow construct.
	PathSelectionRatio	Percent of the selection of a path at a choice construct.
Load balancing	LoadIm	Load imbalance between activity instances of a fork-join construct.
Performance improvement	SpeedupFactor	Speedup factor.
Resource	RedundantProcessing	The time spent to process some work but finally the work is not utilized.

Table 5. Performance metrics at workflow construct level.

time of G, $ElapsedTime(G)$, and the processing time of G, $ProcessingTime(G)$, are defined based on Equation 8 and 9, respectively. Speedup factor of WF G over WF H, $SpeedupFactor(G,H)$, is defined by

$$SpeedupFactor(G,H) = \frac{ProcessingTime(G)}{ProcessingTime(H)} \qquad (11)$$

Let $ProcInRes(R_i)$ be the processing time consumed by resource R_i. Load imbalance at resource R_i, $LoadImRes(R_i)$ is defined by

$$LoadImRes(R_i) = ProcInRes(R_i) - \frac{\sum_{i=1}^{n}(ProcInRes(R_i))}{n} \qquad (12)$$

3.6. Metric Ontology

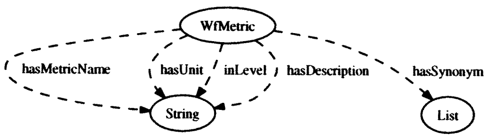

Figure 5. Description of a WF performance metric.

Performance metrics introduced above are described in an ontology named **WfMetricOnto**. A metric is described by class *WfMetric*. Figure 5 presents the concept *WfMetric*. *WfMetric* has five properties: *hasMetricName* specifies the metric name. Property *hasSynonym* specifies other names of the performance metric. Property *hasUnit* specifies the measurement unit of the metric. Property *inLevel* specifies the level with which the metric is associated. Property *hasDescription* explains the performance metric.

4. Ontology for Describing Performance Data of Grid Workflows

We develop an ontology named **WfPerfOnto** for describing performance data of workflows; **WfPerfOnto** is based on OWL [11]. This section just outlines main classes and properties of **WfPerfOnto** shown in Figure 6.

Workflow describes the workflow (WF). A WF has WF constructs (represented by *hasWorkflowConstruct* property), WF graph, etc. A WF construct is described by *WorkflowConstruct*. Each WF construct has activities (*hasActivity*), activity instances (*hasActivityInstance*), WF construct graph, sub WF constructs, etc.

Activity describes an activity of a WF. *ActivityInstance* describes an activity instance. Each *ActivityInstance*, executed on *Resources*, has an execution graph described by class *ExecutionGraph*. An execution graph consists of *ActivityState* and *ActivityEvent* describing activity state and event, respectively. The dependency (control or data) between two activity instances is described by *Dependency*. An activity instance is an object or a subject of a dependency; the object depends on the subject. Activity instances have invoked applications (*hasInvokedApplication*).

InvokedApplication describes an invoked application of an activity. Each *InvokedApplication* is associated with a *SIR* [13], which represents the structure of the application, with a *DRG*, which represents the dynamic code region call graph [17], and with events occurred inside the application.

The dynamic code region call graph, described by *DRG*, consists of region summaries, each stores summary performance measurements of an instrumented code region in a processing unit. A processing unit, described by *Processin-*

Category	Metric Name	Description
Execution time	ElapsedTime	The latency from the time the workflow starts until the time the workflow finishes.
	ProcessingTime	The actually portion of elapsed time that the workflow spends on processing.
	ParTime	The portion of processing time that workflow activities executed in parallel.
	SeqTime	The portion of processing time that workflow activities executed in sequential manner.
Ratio	QueuingRatio	Mean queuing time per elapsed time.
	MeanProcessingTime	Mean processing time per activity.
	MeanQueuingTime	Mean queuing time per activity.
	ResUtilization	Time that a resource spends on processing work per elapsed time of the workflow.
Correlation	NAPerRes	The number of activities executed on a resource.
	ProcInRes	The period of time that a resource spends on processing work.
	LoadImRes	Load imbalance between processing time of resources.

Table 6. Performance metrics at workflow level.

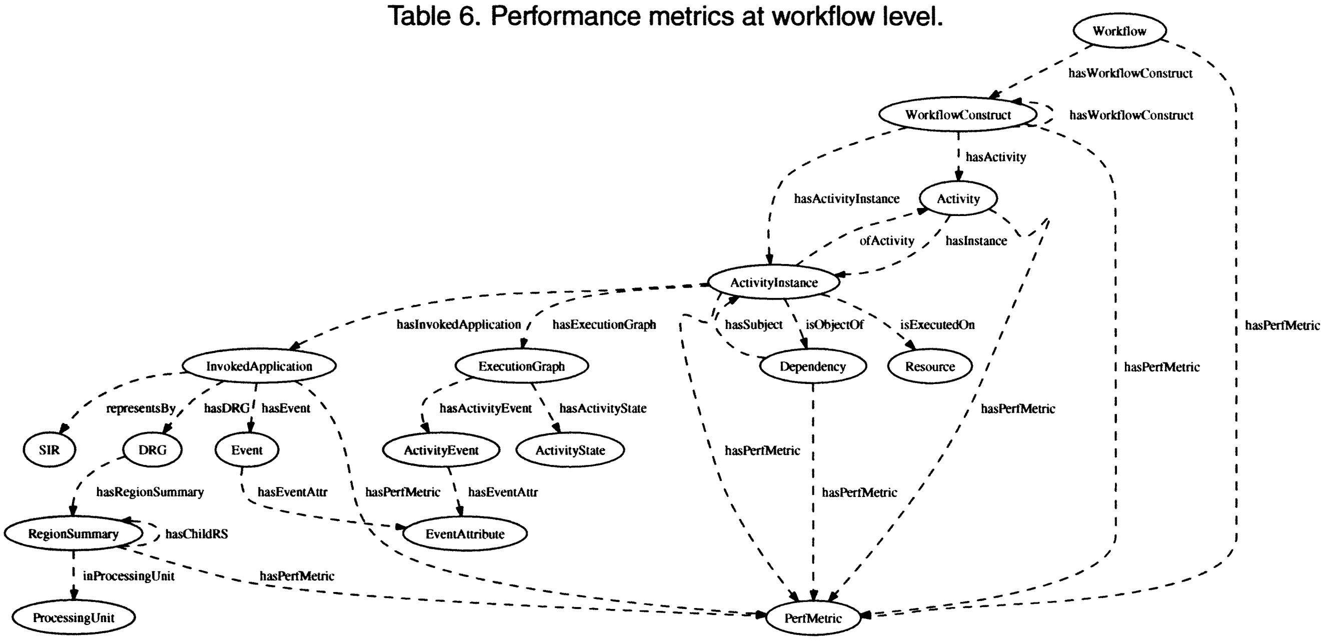

Figure 6. Ontology for describing performance data of workflows.

gUnit, indicates the context in which the code region is executed; the context contains information about the activity identifier, computational node, process identifier and thread identifier. A region summary, described by *RegionSummary* has performance metrics (*hasPerfMetric*) and sub region summaries (*hasChildRS*). *PerfMetric* describes a performance metric, each metric has a name and value. The metric name is in **WFMetricOnto**. *Event* describes an event record. An event happens at a time and has event attributes (*hasEventAttr*). *EventAttribute* describes an attribute of an event that has an attribute name and value.

Performance metrics of *Workflow, WorkflowConstruct, Activity, Dependency, ActivityInstance, InvokedApplication*, and *RegionSummary* are determined through *hasPerfMetric* property.

5. Utilizing WfPerfOnto for Performance Analysis of Grid Workflows

5.1. Describing Performance Data

A performance analysis tool can use **WfPerfOnto** to describe performance data of a workflow. For example, when a client of the performance analysis service requests performance results of a workflow, the client can specify the requests based on **WfPerfOnto** (e.g., by using RDQL [9]). The service can use **WfPerfOnto** to express performance metrics of the workflow. As performance results are described in a well-defined ontology, the client will easily understand and utilize the performance results.

Figure 7 presents an example of a workflow named `Montage`. Dependencies between activities are control dependencies. Figure 8 represents part of the performance data of `Montage` described in **WfPerfOnto**. The performance experiment is executed on two resources. At the top-level, the workflow consists of two workflow constructs, a fork-join construct named `ForkJoin2` and a sequence construct named `Seq`. The fork-join construct can be considered as two sequence constructs named `Seq1ForkJoin2` and `Seq2ForkJoin2`. Activity `mImgtbl2` has two dependencies. Figure 8 presents some interesting performance metrics associated with `mImgtbl2` such as `ElapsedTime` and `SynDelay`.

Although **WfPerfOnto** does not describe (dynamic) monitoring data of resources on which invoked applications of a workflow are executed, from information described in **WfPerfOnto**, e.g., activity events and resource identifiers,

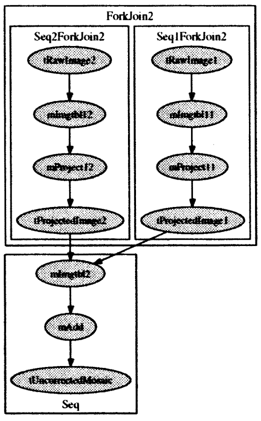

Figure 7. Example of a workflow named `Montage`.

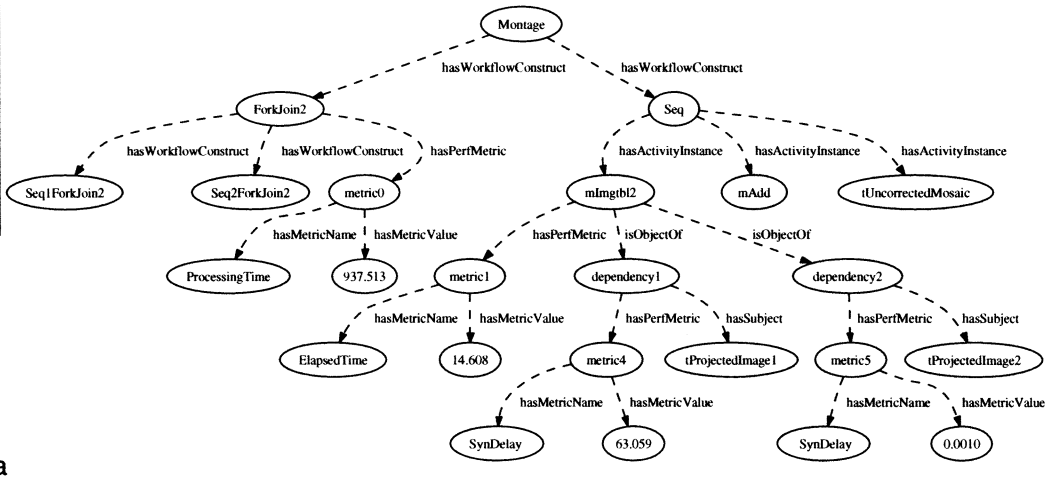

Figure 8. Part of **WfPerfOnto** for workflow `Montage`.

we can obtain (dynamic) monitoring data of resources from infrastructure monitoring services. Thus, we can analyze the performance and dependability of both workflows and resources at the same time.

5.2. Content Language for Analysis Agents

We use **WfPerfOnto** as a content language for distributed agents to share information when they are conducting the performance analysis of workflows in the Grid. In our distributed analysis framework, *analysis agents* are organized into *societies*. Each society has a *major agent* which coordinates the job of agents in its community. Major agents communicate and exchange information each others. Performance analysis requests and performance data exchanged are described by **WfPerfOnto**. Given an analysis request from the client, agents will collaborate in conducting the performance analysis.

Figure 9 presents an example of how agents exchange requests when collaboratively conducting an analysis task. Figure 10 presents an example of a RDQL request for performance analysis. A client sends the request to the major M1 of society S1. From the ActivityID, `Project1`, M1 knows that the request can be processed by the major M3 of society S3 and routes the task. When receiving the request M3 sends the request to agent A3 because M3 knows that A3 can analyze activity `Project1`. A3 conducts the analysis and returns results to major M3 which in turn sends the results to the client. To fulfill a request, an analysis agent may invoke other agents. For example, when A3 has only execution status of activity `Project1`, but in order to compute synchronization delay (metric `SynDelay`), it needs execution status of all activities which `Project1` depends on, therefore, A3 may send other requests to B3 in order to get execution status of other activities.

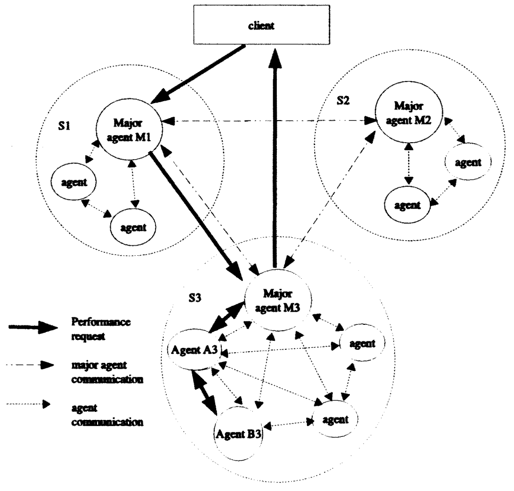

Figure 9. Agents process an analysis request.

```
SELECT ?instance
WHERE  (?instance wfperfonto:ofActivity ?activity)
       (?activity wfperfonto:hasActivityID ''Project1'')
       (?instance wfperfonto:hasPerfMetric ?m)
       (?m wfperfonto:hasMetricName    ''SynDelay'')
USING  wfperfonto FOR <http://dps.uibk.ac.at/wfperfonto#>
```

Figure 10. RDQL query used to request synchronization delay of activity `Project1`.

6. Related Work

Many techniques have been introduced to study quality of service and performance models of workflows, e.g. [7, 4, 6]. Performance metrics in [7, 4] are associated with activities. We consider performance metrics in many levels of detail, e.g. code regions and workflow constructs.

Recently, [10] discusses QoS metrics associated with Grid architecture layers. Our work studies performance metrics of Grid workflows. Existing tools supporting performance analysis of workflows, e.g. [12], have some common performance metrics with our metrics. However, our study covers a large set of performance metrics ranging from the workflow level to the code region level. [16] discusses the role of an ontology of QoS metrics for management Web Services. An ontology for the specification of QoS metrics for tasks and Web services is developed in [3]. However, there is a lack of an ontology for describing performance metrics and performance data of Grid workflows.

Recently, there is a growing effort on mining the workflow [19, 5]. Workflow activities are traced and log information is used to discover the workflow model. Events logged, however, are only at activity level. Workflow mining focuses on discovery workflow model from tracing data where our study is to discuss important performance metrics of workflows and methods to describe performance data of workflows. Workflow event logs can be used to analyze performance metrics proposed by our study.

7. Conclusion and Future Work

The performance and dependability of Grid workflows must be characterized by well-defined performance metrics. This paper presents a novel study of performance metrics of Grid workflows. Performance metrics are associated with multiple levels of abstraction, ranging from a code region to the whole workflow. We have presented an ontology for describing performance data of Grid workflows.

We are currently reevaluating and enhancing the ontology for describing performance data of Grid workflows. Also we are extending the set of performance metrics. We are working on a prototype of a distributed analysis framework in which distributed analysis agents use **WfPerfOnto** based requests to exchange analysis tasks when conducting the performance analysis of Grid workflows.

References

[1] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[2] Junwei Cao, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. Gridflow: Workflow management for grid computing. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 198. IEEE Computer Society, 2003.

[3] Jorge Cardoso and Amit Sheth. Semantic e-workflow composition. *J. Intell. Inf. Syst.*, 21(3):191–225, 2003.

[4] Jorge Cardoso, Amit P. Sheth, and John Miller. Workflow quality of service. In *Proceedings of the IFIP TC5/WG5.12 International Conference on Enterprise Integration and Modeling Technique*, pages 303–311. Kluwer, B.V., 2003.

[5] S. Dustdar, T. Hoffmann, and W.M.P. van der Aalst. Mining of ad-hoc Business Processes with TeamLog. *Data and Knowledge Engineering*, 2005.

[6] Michael C. Jaeger, Gregor Rojec-Goldmann, and Gero Mühl. Qos aggregation for service composition using workflow patterns. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC 2004)*, pages 149–159, Monterey, California, USA, September 2004. IEEE CS Press.

[7] Kwang-Hoon Kim and Clarence A. Ellis. Performance analytic models and analyses for workflow architectures. *Information Systems Frontiers*, 3(3):339–355, 2001.

[8] Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL : A Workflow Framework for Grid Services. Technical report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., July 2002.

[9] RDQL: RDF Data Query Language. http://www.hpl.hp.com/semweb/rdql.htm.

[10] Daniel A. Menasce and Emiliano Casalicchio. Quality of Service Aspects and Metrics in Grid Computing. In *Proc. 2004 Computer Measurement Group Conference*, 2004.

[11] OWL Web Ontology Language Reference. http://www.w3.org/tr/owl-ref/.

[12] Bastin Tony Roy Savarimuthu, Maryam Purvis, and Martin Fleurke. Monitoring and controlling of a multi-agent based workflow system. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 127–132. Australian Computer Society, Inc., 2004.

[13] Clovis Seragiotto, Hong-Linh Truong, Thomas Fahringer, Bernd Mohr, Michael Gerndt, and Tianchao Li. Standardized Intermediate Representation for Fortran, Java, C and C++ Programs. Technical report, Institute for Software Science, University of Vienna, October 2004.

[14] John F. Sowa. *Knowledge Representation: logical, philosophical, and compuational foundations*. Brooks/Cole, Pacific Grove, CA, 2000.

[15] The Condor Team. Dagman (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/.

[16] Vladimir Tosic, Babak Esfandiari, Bernard Pagurek, and Kruti Patel. On requirements for ontologies in management of web services. In *Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web*, pages 237–247. Springer-Verlag, 2002.

[17] Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.

[18] Hong-Linh Truong and Thomas Fahringer. Performance Analysis, Data Sharing and Tools Integration in Grids: New Approach based on Ontology. In *Proceedings of International Conference on Computational Science (ICCS 2004)*, LNCS 3038, pages 424 – 431, Krakow, Poland, Jun 7-9 2004. Springer-Verlag.

[19] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.