# A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures*

Hinde Lilia Bouziane, Christian Pérez, and Thierry Priol

INRIA/IRISA
Campus de Beaulieu — F-35042 Rennes Cedex — France
{Hinde.Bouziane,Christian.Perez,Thierry.Priol}@inria.fr

**Abstract.** Grids are very complex and volatile infrastructures that exhibit parallel and distributed characteristics. To harness their complexity as well as the increasing intricacy of scientific applications, modern software engineering practices are needed. As of today, two major programming models dominate: software component models that are mainly based on a spatial composition and service oriented models with their associated workflow languages promoting a temporal composition. This paper tends to unify these two forms of composition into a coherent spatio-temporal software component model while keeping their benefits. To attest the validity of the proposed approach, we describe how the Grid Component model, as defined by the CoreGRID Network of Excellence, and the Askalon-AGWL workflow language have been adapted.

## 1 Introduction

Grid infrastructures are undoubtedly the most complex computing infrastructures ever built incorporating both parallel and distributed aspects in their implementations. Although they can provide an unprecedented level of performance, designing and implementing scientific applications for Grids represent challenging tasks for programmers. But this is not only due to the intricacy of the infrastructures. Indeed, numerical simulation applications are also becoming more complex involving the coupling of several numerical simulation codes to better simulate physical systems that require a multidisciplinary approach. To cope with the infrastructure and application complexity, it becomes necessary to design scientific applications with modern software engineering practices. Component-based programming or service-oriented programming are good candidates to design these applications using a modular approach. With a component-based approach, an application can be represented as an assembly of software components connected by a set of ports and described using an Architecture Description Language (ADL) while a service-oriented approach tends to represent an application as an orchestration of several services using a workflow language. In some sense, component programming appears as a spatial composition describing the connection between components while service programming promotes a temporal composition expressing the scheduling and the flow of control between services.

* This work was supported by the CoreGRID European Network of Excellence and by the French National Agency for Research project LEGO (ANR-05-CIGC-11).

In the context of Grids, both approaches have been used but in a separate way. In this paper, we show that both spatial and temporal compositions are required in the same programming model. Spatial composition is required to express some specific communication patterns that can be found in multi-physics scientific applications such as in coupled simulation where several simulation codes have to be run simultaneously and have to exchange data at each time step. However, component models do not capture when a given component will communicate with another component it is connected to; consequently all application components have to be deployed in advance on resources and kept until the end of the application. This leads to an inefficient use of resources especially in the context of resource sharing which is one of the aims of the Grid concept. Temporal composition, with respect to resource sharing, is more suitable since the control flow is explicit. It can be used to deploy services only when they are needed allowing thus a better utilization of Grid resources. A programming model, allowing the design of applications using a modular approach, must thus combine spatial and temporal composition to fulfill the programmer's requirements: strong code coupling and efficient use of resources. This paper studies how to combine these two composition schemes together and then it introduces STCM, a spatio-component model, based on the Grid Component Model (GCM) [1] and the ASKALON workflow system [2].

The remainder of this paper is organized as follows. Section 2 introduces and discusses properties of spatial and temporal composition models as well as some related works. Section 3 analyzes some possible designs that combine both compositions into a unique model. In Section 4, we describe STCM, a spatio-temporal component model and an example of application is given in Section 5. Section 6 concludes the paper.

## 2 Composition in Space and Time: Properties and Discussion

This paper focuses on composition as a mean to describe applications' structure. In general, such a structure reflects a reasoning dimension of the programmer. Our interest is focused on two major but orthogonal dimensions: space and time. Reasoning about space or time appears today as a factor separating two programming model trends for building scientific applications: *software component* and *workflow* models. This section presents their respective properties as well as some works attempting to combine them.

### 2.1 Composition in Space

Let us define a spatial composition as a relationship between components if and only if components being involved in the relationship are concurrently active during the time this relationship is valid. In general, components interact through adequate and compatible ports often according to a *provides-uses* paradigm. In most spatial composition models, the direction of the interaction is oriented: it is a user that invokes an operation on a provider. However, the interaction frequency is not specified: it is not known whether the user will actually invoke an operation nor the number of invocations. Thus, components are concurrently active during the time the relation is valid, i.e. the components are connected. Therefore, a spatial composition enables to express the architecture of an application, typically captured by UML **component** diagrams [3]. The spatial

composition principle is followed by most existing component models like CCA [4], CCM [5], FRACTAL [6], SCA [7] and GCM [1], which we briefly present hereafter.

The Grid Component Model or GCM [1] is a component model being specified within the European *CoreGRID* Network of Excellence. It is based on FRACTAL [6], a hierarchical component model, and extends this latter in order to target Grid applications. GCM defines *primitive* and *composite* components. *Composite* components may contain several (sub-)components that form its *content* as illustrated in Figure 1. GCM defines also *controllers* to separate non-functional concerns from the computation ones. In particular, *controllers* are used to manage sub-components. GCM supports several kinds of ports such as RMI or data streaming. GCM provides also an Architecture Description Language (ADL) which allows the specification of both components and their composition in a same phase.

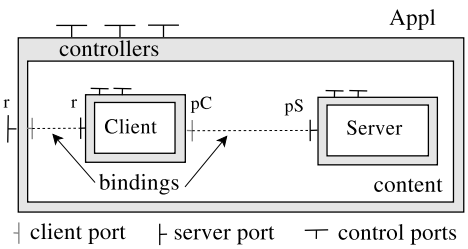



**Fig. 1:** Example of a GCM component.

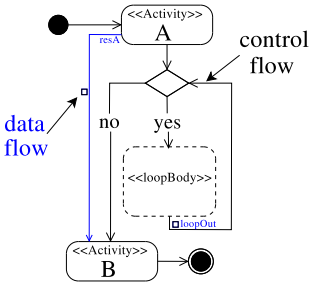
### 2.2 Composition in Time

A temporal composition can be defined as a relationship between tasks if and only if it expresses an execution order of the tasks. There are two classical formalisms for describing such a relationship: data flows and control flows. Data flows focus on the dependencies coming from data availability: the outputs of some tasks $t_i$ are inputs of a task $T$. The execution of $T$ depends on that of all $t_i$. In control flows, the execution order is given by some control constructs such as sequences, branches or loops. Temporal compositions enable expression of the sequence of actions which typically may be captured by UML **activity** diagrams [3]. There exist many environments [8] that deal with temporal compositions such as workflow systems like ASKALON [2], TRIANA [9], KEPLER [10], BPEL4WS [11], etc. For this paper, let us focus on ASKALON-AGWL.

ASKALON [12] is a Grid environment dedicated to the development and execution of scientific applications, being developed at the University of Innsbruck, Austria. It proposes the *Abstract Grid Workflow Language* (AGWL) [2]. This language is viewed by the designer under an *UML* activity diagram formalism. It offers a hierarchical model made of atomic and composite activities (sub-workflows). A composition is done with respect to both data flow and control flow compositions, as illustrated in Figure 2. A data flow is specified by simply connecting input data port to output data port of dependent activities, while the control flow describes the execution order of activities. AGWL supports several control structures like sequences, branches (*if* and *switch*), loops (*for* and *while*) and parallel structures (*parallelFor* and *parallelForEach*), etc.

### 2.3 Discussion

Spatial composition is well suited to describe components that must co-exist simultaneously and may communicate. It is the case for strong code coupling simulations such as meteorological simulations. The main limitation of spatial compositions is they do




```
<agwl-workflow>
 <importATD url=".../activTypes.atd" name="Appl"/>
  ...
  <activity name="A" type="Appl:typeA">
   <dataIn name="dinA" source="dinApplExample"/>
   <dataOut name="resA" />
  </activity>
  <while name="loop">
   <dataIn name="dinL" loopSource="loopOut"/>
   <value>true</value>
   <condition>loopOut='true'</condition>
   <loopBody> <!-- some activities -->
    <dataOut name="loopOut" />
   </loopBody>
   </while>
  <activity name="B" type="Appl:typeB">
    <dataIn name="dinB" source="resA"/> ...
  </activity> ...
</agwl-workflow>
```

**Fig. 2.** A composition example in ASKALON-AGWL.

not explicitly capture the temporal dimension. That may lead to an underutilization of the resources because of an overestimation of needed ones. It is possible to embed an orchestration into a component driver. However, any modification on the application structure requires to modify the code. Lazy component instantiation also does not fully solve the problem as it is not known when a component can be safely destroyed.

Temporal composition is able to capture the temporal dimension and hence it enables efficient resource management. Nevertheless, its main limitation is the lack of support to express that two running tasks must communicate, as for example strong code coupling simulations. The solution of externalizing the loop of a code limits the coupling to coarse grained codes with respect to the overhead of launching a task.

**Attempts to Merge Spatial and Temporal Compositions** To capture the good properties of the two models, some solutions have been proposed. ICENI [13] describes the internal behavior of a component with a workflow formalism. That helps to compute an optimized *spatial* deployment plan. However, it does not capture temporal relationship *between* components. Workflow models like in TRIANA or ASKALON enable spatial compositions. However, they are often hidden in tasks' implementation. As far as we know, workflow engines are not aware of underlying spatial compositions. Thus, models like in [14] propose specialized tasks dedicated to communications between *communicating* processes. However, that requires to modify codes to extract communications.

To summarize, the limitations seem to mainly come from the fact that the spatial and temporal dimensions are handled at distinct levels of the application structure. Hence, this paper focuses on a model where the two dimensions can co-exist at a same level.

## 3 Toward a Spatio-Temporal Composition Model

### 3.1 Targeted Properties

Our goal is to define a model that enables the concurrent use of both spatial and temporal composition paradigms at any level of an application structure. First, the model

should provide a quite high level of abstraction. In particular, it should abstract the resource infrastructures so that the Grid remains invisible from the programmer point of view. Second, the composition model should be rich enough to support a wide range of composition paradigms like control flow constructs (sequence, conditions, loops, etc.), method invocation, message passing, etc. Third, supporting many kinds of composition paradigms may lead to a complex life-cycle management. Hence, the model should offer a simple life-cycle model for combined spatial and temporal compositions so that the behavior of a whole application is quite easy to determine. Fourth, the model should be *hierarchical* and should provide all composition paradigms at any level of a hierarchy. Hierarchy appears as an important property to structure applications and to improve reusability. Fifth, as we aim at leveraging existing works, it should be possible to specify the model as an extension of some existing ones.

### 3.2 Analysis of Design Models for a Spatio-Temporal Composition Model

Defining a spatio-temporal composition model requires to instantiate the concepts encountered in Section 2 in a coherent model. This section analyzes some design approaches keeping in mind the properties presented in Section 3.1.

There are two kinds of entities that may be embedded into a code: components and tasks. From an architectural point of view, they are very similar: they are black boxes with some communication ports. The main difference is on their life-cycle: a task is implicitly instantiated only at the time of its execution. Hence, we fuse them into the term task-component, which we define as a component supporting the concept of task. Hence, a mechanism is needed to define input and output ports and to bind tasks to components. The term task-component is used to distinguish between components supporting tasks and classical ones. It is just a notation as task-components are components.

As we start from a component model, the concept of ports keeps its usual definition. Spatial composition is thus directly inherited. However, the concept of port has to be extended with input/output ports for temporal compositions. As it consists in associating a piece of data to a port, the basic mechanism looks very similar to event ports.

A third issue is to define the rules governing task-component life-cycle. Such rules should state when a component can and/or must be created/destroyed. For example, the life-cycle of a task-component with only input and output ports can be controlled by its temporal relationship: it can be instantiated when its inputs are ready and destroyed when outputs have been retrieved. However, rules become more complex when a task-component has temporal and spatial ports.

Basing a spatio-temporal composition model on a data flow model is quite straightforward. The composition of input and output ports following the same philosophy as spatial ports, i.e. connections of compatible ports, it seems possible to slightly extend assembly languages of component models – like GCM ADL – to take them into account into an assembly with data flow compositions representing temporal compositions.

It seems also possible to integrate a control flow model. Control flow models are based on “programmable” constructions while component assemblies are based on description languages. Hence, an issue is to deal with the instructions of such a programmable language. There are two classical approaches. The first approach embeds every element of the language into a component, like in TRIANA, which provides a

**Table 1.** GCM and AGWL concepts reused for defining a spatio-temporal model.

| Required concept | Provided concepts | Selected strategy |
|---|---|---|
| Task-Component | provided, used operations and tasks | extend GCM with task concept |
| Ports | spatial: GCM ports<br>temporal: input and output data | extend GCM with<br>temporal ports |
| Composition | spatial: GCM bindings<br>temporal: data and control flow: AGWL | extend AGWL with GCM<br>components and spatial bindings |
| Component life-cycle | states and transitions | inferred from composition |

model that is easily extensible by adding new components. However, as components embed the control flow, it turns out that the control flow of the application is not visible: it may restrict optimizations like advance reservation of resources unless using behavioral component models. The second approach distinguishes language instructions from user components, like in many workflow languages. It limits language extensions but it enables runtime optimizations as the language is known.

## 4 STCM: A Spatio-Temporal Model Based on GCM and AGWL

This section presents STCM, a spatio-temporal model based on both GCM and AGWL as well as the objectives presented in Section 3. In particular, the proposal is based on choosing, reusing and potentially merging or extending the specification of components, ports, tasks and the composition model offered by GCM and/or AGWL. Our choices are essentially motivated by keeping the advantages of each model. Table 1 sums up our strategy to reuse GCM and AGWL principal concepts in order to define a spatio-temporal model. The remainder of this section reviews these points in more depth.

### 4.1 Extending GCM Components with Tasks and Temporal Ports

The type of a component being defined by its ports, a new family of ports is required to define a task-component. Let us call them input and output ports. In contrast to classical client/server ports, that provide a method call semantic, input/output ports are attached to a data type. Hence, STCM provides typed input and output ports. They are provided through an extension of the GCM *TypeFactory* interface dedicated to create types. A *createFcTemporalType* operation creates the definition of an input (*isInput* = *true*) or output (*isInput* = *false*) port named *name* and for which the type is determined by a *data type* argument. As temporal ports are distinguished from classical ones, a component type declaration is also extended to include this new kind of ports.

The next step is to support a task within a task-component. A task can be viewed as a particular operation to be implemented by a user. The definition of such an operation depends on several assumptions. For example, multi-task components required to define a triplet (task, inputs, outputs) for each task, while it may be implicit for single task-component. Because of lack of space, the support of only one task per component is presented here. A task-component is a component which implements a *TaskController* interface which contains only a `void task()` operation which is called when the task needs to be executed. Input data are retrieved through input ports (through getter-like operations) and output data are set through output ports (through setter-like operations).

### 4.2 Life Cycle Management of Task-Components

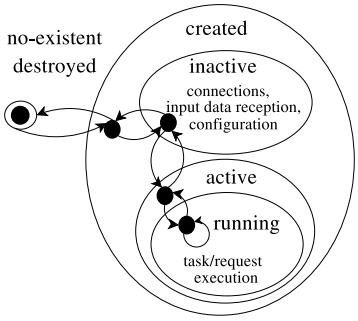



**Fig. 3:** State diagram.

Figure 3 presents a proposed state machine diagram with respect to the life-cycle of task-components. Compared to a classical task, where its activation corresponds to its execution, the active state of a task-component may be longer than the task running duration. The duration of the active state depends mainly on both the temporal composition and the requirement of the presence of provided functionality by a component. Hence, a component can be active without any running task like a standard component.

### 4.3 A Composition Language Based on a Modified AGWL

The STCM composition model is inspired from the AGWL language. The objective is to preserve its algorithmic composition logic but based on a task-component assembly view. Hence, the approach is essentially based on the replacement of the activity concept by a task-component one. Figure 4 presents the main elements of the grammar of the STCM language. Component definition looks like in GCM ADL but with the support of temporal ports as well as the possibility to connect them when being defined. Moreover, the language has dedicated instructions (`setPort` and `unsetPort`) to connect/disconnect ports.

As in AGWL, control flow composition is expressed as the content of composites. Then, it is straightforward to adapt all AGWL control flow constructions. Such instructions can be seen as pre-defined components with a known internal behavior. In STCM ADL, a component instance can be defined in the *declaration* part of a composite assembly or a control flow instruction. It results in distinct behaviors: the former aligns the instance creation and destruction with the composite ones, while the latter enables a dynamic creation and destruction.

The semantics of such a language has yet to be defined as for example with respect to when a component instance can be safely destroyed. We are working on the definition of a semantics able to reflect as much as possible a behavior based on a simple priority system: *if a spatial connection is specified within a control structure body then the temporal dimension is prevailing, otherwise the spatial dimension is to be considered first.*

### 4.4 Proof-of-Concept Implementation

In order to test the feasibility of the model, we have implemented a proof-of-concept interpreter of the STCM language based on the ANTLR language tool. The interpreter parses the language and generates calls to a GCM extended API so as to manage components, like component creation/destruction, port connection, as well as task invocation. It does not yet support all control flow instructions. A full implementation of the model requires to define a semantic and to implement/adapt a workflow engine.

```
component    ::= <component name=string (extends=string)?>
                  port* content? membrane?
                 </component>
port         ::= clientport | serverport | inport | outport | attribute
clientport   ::= <clientPort name=string type=string (set=string)?/>
serverport   ::= <serverPort name=string type=string/>
inport       ::= <dataIn  name=string type=string (set=string)?/>
outport      ::= <dataOut  name=string type=string/>
attribute    ::= <attribute name=string type=string (set=string)?/>

membrane     ::= <controllerDesc desc=string/>
content      ::= primitive | composite
primitive    ::= <impl type=string signature=string/>
composite    ::= <body> stcmassembly </body>
stcmassembly ::= declaration? instruction?

declaration  ::= <declare> component* instance* configport* </declare>
instance     ::= <instance name=string componentRef=string>
                   content? membrane?
                 </instance>
configport   ::= clientserver | inout
clientserver ::= <setPort client=string server=string/>
                 | <unsetPort client=string (server=string)?/>
inout        ::= <setPort in=string out=string/>
                 | <unsetPort in=string (out=string)?/>

instruction  ::= instance | executetask | configport | seq | if | switch | while
                 | for | forEach | dag | parallel | parallelFor | parallelForEach
executetask  ::= <exectask nameInstance=string/>

seq          ::= <sequence name=string>port* declaration instruction+</sequence>

if           ::= <if name=string> port* declaration condition then else?</if>
condition    ::= <condition> expr </condition>
then         ::= <then> stcmassembly </then>
else         ::= <else> stcmassembly </else>

parallel     ::= <parallel name=string> port* declaration section+</parallel>
section      ::= <section> stcmassembly </section>

switch       ::= <switch name=string> port* declaration case+ default?</switch>
case         ::= <case condition=string (break=boolean)?> stcmassembly </case>
default      ::= <default> stcmassembly </default>
boolean      ::= true | false

// Same principle for while, for, forEach, dag, parallelFor and parallelForEach.
// expr represents a logical expression as in AGWL , with the same restrictions.
```

**Fig. 4.** Overview of the STCM grammar. Keywords are in bold, while strings are in italic.

## 5 Example of an Application Description

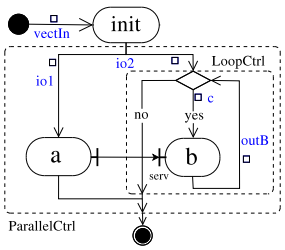



**Fig. 5:** Application example.

Figure 5 illustrates a simplified STCM application coming from the French ANR LEGO project. This application contains two coupled codes represented by the spatially connected components `a` and `b`. Component `a` operates on a matrix, initialized according to some initial conditions defined by Component `init`. The result computed by Component `a` depends on data provided by Component `b`, data which depend on the iterative convergence computation which Component `b` is involved in. It

```
1  <component name ="exApp">                  18 <parallel name="ParallelCtrl">
2   <dataIn name="vectIn"  type="Vect"/>      19  <instance name="a" compRef="A"/>
3   <body><component name="Init">            20  <instance name="b" compRef="B"/>
4     <dataIn  name="ii1" ... set="vectIn"/> 21  <section>
5     <dataOut name="io1" ... />             22   <exectask nameInstance="a"/>
6     <dataOut name="io2" type="double" />   23  </section>
7    </component>                            24  <section>
8    <component name="A">                    25   <while name="LoopCtrl">
9     <dataIn name="inA" ... set="init.io1"/>26    <dataIn name="c" type="double"
10    <clientPort name="pA" ... set="B.pB"/> 27            set="init.io2"
11   </component>                            28            loopSet="B.outB"/>
12   <component name="B">                    29    <condition> c<0.1 </condition>
13    // in: double inB, out: double outB    30    <loopBody>
14    <serverPort name="pB" type="GetRes"/>  31     <exectask nameInstance="b">
15   </component>                            32      <dataIn name="inB" set="c"/>
16   <sequence name="seq1">                  33     </component>
17    <instance name="init" compRef="Init"   34    </loopBody>
      // lines 18-37 are on the right        35   </while>
38   </sequence></body>                      36  </section>
39 </assembly>                               37 </parallel>
```

**Fig. 6.** Main elements of an application description in STCM.

is expected that Component b has a persistent active state for continuous a requesting. Therefore, the integration of b in the loop has to preserve the first created instance during all iterations. For simplicity, the detailed structure of GCM components (membranes, contents, implementations) are not represented in this section.
Figure 6 shows how this application can be expressed with the STCM language. The expressed execution ordering matched perfectly with the specified requirements. In particular, Instance b of Component B is declared in the header of the parallel section. Hence, it is not destroyed at each iteration of the while loop.

While STCM offers means to explicitly express the specified behavior by the assembly, it is not the case when using separately GCM or AGWL. With STCM, it is usual to hide the temporal logic in a driver component. Depending on the programmer expertise, this component can manage the life-cycle of `init`, a and b. This management is required to avoid overconsumption of resources, in particular if components are composite/parallel. However, that may be complex to be done by the user. With AGWL, there is no mean to express the spatial dependence between a and b, which is usually hidden in tasks' code. That may limit reusability. In addition, the stateless property of tasks implies the b's state to be saved/reloaded for each iteration. That may lead to inefficient execution. On the contrary, STCM offers a more powerful assembly model in term of behavior expressiveness. This is relevant to ease programing, improve reusability, enable automatic management of an assembly and optimize resources usage.

## 6 Conclusion and Future Works

In order to harness the programmability of Grids, two major approaches are used to develop applications: software component models mainly used by strongly coupled applications and workflow models mainly used by loosely coupled applications. As both models have benefits and drawbacks with respect to some algorithmic patterns, this paper explores the possibility of designing a model that support both composition models. The paper has analyzed some designs for combining both of them. As a result, the paper

describes STCM, a spatio-temporal component model based on two existing models – GCM and ASKALON. Some benefits has been shown through an example.

Future works consist in defining a semantic for the STCM language as well as having a full implementation, either based on a new workflow engine, on the adaptation of an existing one, or on the compilation of STCM to plain AGWL. Though the latter should not lead to the best implementation, it may be enough to validate STCM.

## References


1. Institute, P.M.: Basic features of the grid component model. CoreGRID Delivrable D.PM.04, CoreGRID (march 2007)
2. Fahringer, T., Qin, J., Hainzer, S.: Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and Grid 2005 (CCGrid 2005). Volume 2., Cardiff, UK (May 2005) 676–685
3. OMG: Unified modeling language. Document formal/2007-02-05 (February 2007)
4. Bernholdt, D.E., Allan, B.A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A component architecture for high-performance scientific computing. International Journal of High Performance Computing Applications **20**(2) (2006) 163–202
5. OMG: CORBA component model, v4.0. Document formal/2006-04-01 (April 2006)
6. Bruneton, E., Coupaye, T., Stefani, J.: The Fractal Component Model, version 2.0-3. Technical report, ObjectWeb consortium, (February 2004)
7. Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O., Ielceanu, S., Miller, A., Karmarkar, A., Malhotra, A., Marino, J., Nally, M., Newcomer, E., Patil, S., Pavlik, G., Raepple, M., Rowley, M., Tam, K., Vorthmann, S., Walker, P., Waterman, L.: SCA Service Component Architecture - Assembly Model Specification, version 1.0. Technical report, Open Service Oriented Architecture collaboration (OSOA) (March 2007)
8. Yu, J., Buyya, R.: A taxonomy of workflow management systems for grid computing. Journal of Grid Computing **3**(3-4) (september 2005) 171–200
9. Taylor, I., Shields, M., Wang, I., Harrison, A.: Visual Grid Workflow in Triana. Journal of Grid Computing **3**(3-4) (September 2005) 153–169
10. Altintas, I., Birnbaum, A., Baldridge, K.K., Sudholt, W., Miller, M., Amoreira, C., Yohann: A framework for the design and reuse of grid workflows. In: First Intl. Workshop on Scientific Applications of Grid Computing (SAG'04)), Berlin/Heidelberg, Springer (2005) 120–133
11. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1. Technical report (May 2003)
12. Thomas, F., Radu, P., Rubing, D., Francesco, N., Stefan, P., Jun, Q., Mumtaz, S., Hong-Linh, T., Alex, V., Marek, W.: ASKALON: A Grid Application Development and Computing Environment. In: Proceedings of the 6th International Workshop on Grid Computing, Seattle, USA (November 2005) 122–131
13. Furmento, N., Mayer, A., McGough, S., Newhouse, S., Field, T., Darlington, J.: ICENI: Optimisation of component applications within a grid environment. Journal of Parallel Computing **28**(12) (2002) 1753–1772
14. Pllana, S., Fahringer, T.: Uml based modeling of performance oriented parallel and distributed applications. In Yucesan, E., Chen, C.H., Snowdon, J., Charnes, J., eds.: Proc. of the 2002 Winter Simulation Conference, San Diego, California, USA, IEEE (December 2002)