Combining a Software Component Model and a Workflow Language into a Component Model with Spatial and Temporal Compositions

Hinde Lilia Bouziane, Christian Pérez, Thierry Priol

Combining a Software Component Model and a Workflow Language into a Component Model with Spatial and Temporal Compositions

Hinde Bouziane — Christian Pérez — Thierry Priol

N° 6421

Janvier 2008

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6421--FR+ENG

Combining a Software Component Model and a Workflow Language into a Component Model with Spatial and Temporal Compositions

Hinde Bouziane , Christian Pérez , Thierry Priol

Thème NUM — Systèmes numériques
Équipes-Projets PARIS

Rapport de recherche n° 6421 — Janvier 2008 — 20 pages

Abstract: Grids are very complex and volatile infrastructures that exhibit parallel and distributed aspects. To harness its complexity as well as the increasing intricacy of scientific applications, modern software engineering practices are needed. As of today, two major models dominate: software component models that are mainly based on spatial compositions and service oriented models with their associated workflow languages promoting temporal compositions. This paper unifies 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.

Key-words: Software components, workflow models, spatial composition, temporal composition, Grids

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).

Un modèle de composants offrant des compositions spatials et temporels basé sur un modèle de composants logiciels et un langage de workflow

Résumé : Les grilles de calcul sont des infrastructures complexes et volatiles qui exhibent à la fois des aspects parallèles et distribués. Afin de maîtriser leur complexité ainsi que celle toujours croissante des applications scientifiques, des solutions modernes du génie logiciel apparaissent pertinentes. Actuellement, deux modèles majeurs dominent : les modèles de composants logiciels qui sont principalement basés sur des compositions spatiales et les modèles orienté services avec leur langage de workflow qui privilégient des compositions temporelles. Ce papier unifie ces deux formes de compositions dans un modèle cohérent de composants logiciels spatio-temporel en préservant les avantages respectifs des deux approches. Pour valider notre approche, nous décrivons comment le modèle de composant GCM, défini par le réseau d'excellence CoreGRID, et le langage de workflow AGWL de l'environnement Askalon peuvent été adaptés.

Mots-clés : Composants logiciels, modèles de workflow, composition spatiale, composition temporelle, grille de calcul

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 a challenging task for the 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 multi-disciplinary 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 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 connexion between components while service programming promotes a temporal composition expressing the scheduling and the flow of control between services.

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, spatial composition fails to specify resource dependencies because component behavior is hidden in its implementation. It is thus not possible to know when a given component will communicate with another component it is connected to. Since spatial composition do not express the control flow between components, all application components have to be deployed in advance on resources even if some of the components will not be called immediately. This leads to an inefficient use of resources especially in the context of resource sharing which is one of the aim 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 Grid programming model, allowing the design of applications using a modular approach, must thus combine spatial and temporal composition. The objective of this paper is to show how to combine these two composition schemes together and to provide a concrete model based on the Grid Component Model (GCM) and the ASKALON workflow system.

The remainder of this paper is organized as follows. In Section 2, we introduce and discuss properties of spatial and temporal composition models as well as some related works. Section 3 analyzes some possible designs that combines both compositions into a unique model. Then, a concrete model is presented in Section 4 and an example is given in Section 5. Section 6 concludes the paper.

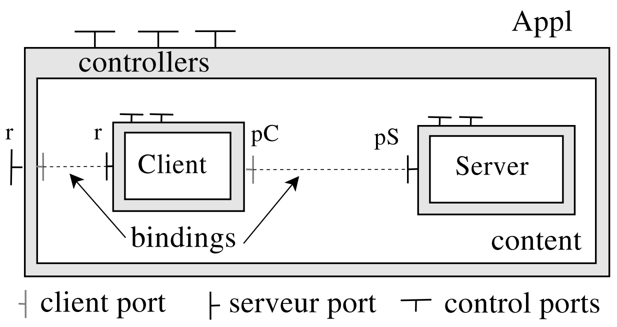


Figure 1: A GCM component.

2 Composition in space and time: properties and discussion

The composition issue addressed in this paper is on the concepts an application's structure is built onto. 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. Models which are of particular interest in this paper are also introduced.

2.1 Composition in space

Let 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 ports, according for instance to the provides-uses paradigm.

Hence, components must have adequate and compatible ports to be composed. For example, for a provides-uses composition, a component acts as a provider (resp. a user) if and only if it exhibits a `provides` (resp. `uses`) port. Then, the composition determines the direction of allowed interaction. In most spatial composition models, the direction is oriented: it is the user that invokes an operation on a provider. However, they do not inform on the interaction frequency: it is not known whether the user will actually invoke an operation neither the number of invocations. It is only known that the components concurrently exist during the time the relation is valid, i.e. the components are connected. Consequently, a spatial composition enables to express the architecture of an application, typically captured by UML component diagrams[11]. Spatial composition principle is followed by most proposed component models like CCA [3], CCM [10], FRACTAL [5], GCM [8], SCA [9], SOFA [4], etc. Let present GCM more in depth.

The Grid Component Model or GCM [8] is an ongoing component model being specified within the European *CoreGRID* Network of Excellence. It is based on FRACTAL [5], a hierarchical component model, and extends this latter

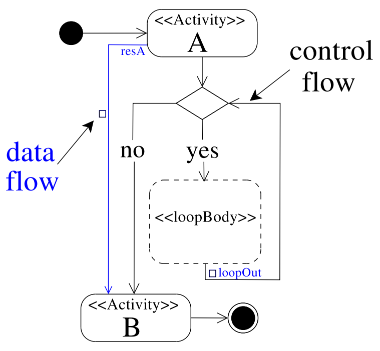
```
ComponentType ClientT = tf.createFcType(new InterfaceType[]{
 tf.createFcItfType("pC", "Compute", true,..), // true = client port
 tf.createFcItfType("r", "Run", false,..)});  // false = server port
...
Component Appl = gf.newFcInstance(ApplT, "composite", null);
Component Client = gf.newFcInstance(ClientT, "primitive", "CImpl");
...
ContentController cc = Appl.getFcInterface("content-controller");
cc.addFcSubComponent(Client);
cc.addFcSubComponent(Server);
...
((BindingController)Client.getFcInterface("binding-controller"))
 .bindFc("pC", Server.getFcInterface("pS"));
```

Figure 2: Example of a composite definition based on GCM API.

in order to target Grid applications. GCM is a hierarchical model which defines *primitive* and *composite* components. A *composite* component, as described in Figure 1, may contain several (sub-)components that form its *content.* GCM defines also *controllers* to separate non-functional concerns from the computation ones. In particular, *controllers* are used to manage sub-components, in particular their life-cycle (connections, creation, etc). GCM supports several kinds of ports such as RPC/RMI, data streaming, events, data sharing, etc. Parallel communications are also supported through *1-to-n*, *n-to-1* or *n-to-m* patterns. Supporting several communication paradigms allows a designer to simply express many kinds of interaction semantics as well as to relieve him/her of the burden of implementing underlying mechanisms such as data sharing services (for data ports) or event channels (for event ports). GCM provides an *API* that defines operations for specifying component types, creating, connecting/disconnecting component instances, as well as operations for introspection and controller specifications, etc. Figure 2 gives an overview of the GCM *API* usage to describe part of the assembly associated to the composite component of Figure 1. GCM provides also an Architecture Description Language (ADL) which allows the specification of both components and their composition in a same phase.

2.2 Composition in time

Let define a temporal composition 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 kind of relationship: data flows or control flows. A data flow focuses 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 the one of all t_i. In control flow models, the execution order of tasks is given by some instructions. Most existing models provide instructions such as sequence, branches, loops, etc. Temporal compositions enable expressing the sequence of actions within an application which typically may be captured by UML activity diagrams[11]. There exist many environments that deals with temporal compositions such as workflow systems like Askalon [6], Triana [13], Kepler [1], BPEL4WS [2]



```
<agwl-workflow>
 <importATD ... name="Appl"/>
  ...
  <activity name="A" type="Appl:typeA">
   <dataIn name="d_A" source="dinAppl"/>
   <dataOut name="resA" />
  </activity>
  <while name="l1">
   <dataIn name="d_L" loopSource="l1Out"/>
   <value>true</value>
   <condition>loopOut='true'</condition>
   <loopBody> <!-- some activities -->
    <dataOut name="l1Out" />
   </loopBody>
   </while>
  <activity name="B" type="Appl:typeB">
    <dataIn name="d_B" source="resA"/> ...
  </activity> ...
</agwl-workflow>
```

Figure 3: A composition example in ASKALON-AGWL.

or other cited in a taxonomy of Grid workflow systems provided in [15]. For this paper, let introduce ASKALON-AGWL as an example.

ASKALON [14] 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) [6] to describe an application. This language, is viewed by the designer under an *UML* activity diagram formalism. AGWL enables the separation of functional concerns of tasks (designed by the term *activities*) from their potential resource dependencies, which may be specified separately by high-level constraints. AGWL offers a hierarchical composition model. An activity can be atomic or composite. A composite activity is a sub-workflow which can be itself composed of atomic activities and/or sub-workflows. A composition is done with respect to both data flow and control flow compositions, as described in Figure 3. A data flow composition is specified by simply connecting input data to output data 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

In some cases, spatial composition is very useful to describe the fact that some components must co-exist simultaneously and that they must exchange functionalities. It is the case for strong code coupling simulations like for example meteorological simulations.

The main limitation of spatial compositions is that they do not explicitly capture the temporal dimension. As far as we know, ADL based component models are not able, through an assembly description, to express the fact that two components `A` and `B` do not need to be instantiated simultaneously because for example `A` is a pre-processing with respect to `B`. From the ADL point of view, all components need to be instantiated during the application lifetime. It may lead to an underutilization of resources because of an overestimation of needed resources. Using an API to dynamically create/destroy components as can be done for example in CCA or GCM, partially solves the problem. A driver component can orchestrate components creation/connection/destruction. However, the drawback is that the composition is hidden in the code. Hence, any modification on the application structure requires to modify the code.

As for spatial composition, temporal composition improves code reuse by assembling black box. However, in contrast to spatial composition models, its main advantage is the enabling of efficient resources management thanks to the expressiveness of temporal dependencies. Nevertheless, the main limitation of temporal composition 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.

2.4 Some attempts for combining spatial and temporal compositions

According to the previous discussion, spatial and temporal composition models appear to be complementary: the drawback of each one is an advantage of the other. As an attempt to capture the good properties of the two models, some efforts can be found in the literature.

ICENI [7] addresses the scheduling issue of a spatial composition. The proposed approach is based on a meta-data principle. It consists in describing the internal behavior of a component in a workflow formalism. Such a description is to be considered by an ICENI tool to help the decision of a deployment plan. Even if this approach may results to an optimized plan, the benefit is only partial. The algorithmic logic of an application remains hidden in the code. In addition, producing meta-data requires to know implementation details. ICENI supposes that such informations are produced by the component's developer. A considerable effort is then required when legacy codes are aimed to be reused. At the same time, there is no guarantee that the meta-data is conformed to the developed code.

In contrast with a meta-data approach, workflow models like Triana [13], Askalon [6], etc. enable a spatial composition to be re-used. However, it is generally hidden in a task implementation and is embedded in an executable format. As far as we know, a workflow engine is not aware about the underlying spatial composition. Hence, the responsibility of its deployment is left to the user. Therefore, it may become very complex to realize an efficient resources usage for the whole application.

Some workflow models like in [12] use another approach that consists in defining particular tasks dedicated to message passing and remote method invocations, where communicating processes have to be determined. The main difficulty is that it requires to modify codes so as to extract communication concerns. Compared to a component programming approach, it seems very complex for the development of large applications.

To summarize, the limitation of presented approaches seems to essentially 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 let a developer design applications by offering him/her a model that enables the concurrent use of both spatial and temporal composition paradigms at any level of an application structure. This section starts examining the properties we expect such a model should offer.

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. Hence, an application can be executed everywhere. Moreover, the model has to be expressive enough to allow applications to be adapted to any kind of infrastructures. Second, the composition model should be rich enough to support a wide range of composition paradigms like control flow structures (sequence, conditions, loops, etc.), method invocation, message passing, etc. The possibility of expressing such paradigms when composing an application will offer an easy and a natural way of programming. 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 support all composition paradigms at any level of a hierarchy. Hierarchy appears as an important property to structure applications and to improve re-usability. 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.

Task-Component There are two kinds of entities that embed a code: components and tasks. In order to have a single basic composition concept, we aim at fusion them. From an architectural point of view, a task is very similar to a component: it is a black box with some ports. The main difference is on their life-cycle: a task is implicitly instantiated only the time of its computation. Hence, we have chosen the component concept as the basic foundation and have imagined a mechanism such that tasks can be defined inside a component. Moreover, single and multiple task per component models can be easily supported. In a single task model, the inputs (resp. outputs) of a task are all the input (resp. output) ports (introduced hereinafter) of its component. To support a multi-task model, a mechanism is needed to bind each task to a subset of the input and output ports. Subsequently, the term task-component is going to be used to distinguish between components supporting tasks and classical ones. It is just a notation as task-components are components.

Life-cycle A second issue is to define the rules that govern the life-cycle of task-components. Such rules should state when a component can and/or must be created/destroyed. For example, the life-cycle of a task-component that has 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 stored. However, if a task-component has some spatial ports, the rules become more complex. A task-component has to be instantiated as long as there is another component that may need to access some of its spatial ports.

Spatial composition model and ports As we have decided to start from a component model, the concept of ports keeps its definition. Spatial composition is also inherited. However, the concept of port has to be extended with input and output ports to support temporal composition. As it consists in associating a piece of data to a port, the basic mechanism looks like very similar to event ports. Hence, it seems quite easy to add such ports to component models.

Data flow temporal composition model Basing a spatio-temporal composition model on a data flow model is quite straightforward. As the composition of input and output ports follows the same philosophy as spatial ports, that is to say based on a connection of compatible ports, it seems possible to slightly extend assembly languages of component models – like CCM ADL or GCM ADL – to take them into account and resulting to an assembly in which a constructed data flow reflects a temporal composition.

Control flow temporal composition model It is also possible to integrate a control flow model which usually provides a more familiar model for programmers. However, control flow models are based on "programmable" languages 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: to let every instruction of the language be hidden in a component or to let some instructions such as conditions or loops be special instructions. Let analyze these two options.

Considering every element of the language hidden in components (Triana like approach) provides a simple model that is easily extensible by adding new components. However, as components embed the control flow and components are black-boxes, it turns out that the control flow of the application is not visible: it may restrict optimizations like advance reservation of resources. It may be possible to extend task-components with meta-data to describe their behavior but it will make the model more complex: the definition of task-components appears not straightforward as the control and data flow need to be transfered: another kind of ports is needed to capture the control part.

Providing the same expressiveness as common workflow languages requires to distinguish user level task-components from language instructions. Hence, a specific runtime is required to execute these instructions. It leads to a more complex runtime than for the previous model, but the complexity is not so high:

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

Figure 4: GCM and AGWL concepts used for defining a spatio-temporal model.

such a runtime looks like a workflow engine. An open issue is to study how such an engine can cohabit with component frameworks.

We select the second option because it is what users expect from current workflow models. Then, task-components can be composed as in the data-flow model – thanks to data input and output ports – as the control is managed outside components by the instructions of the language.

4 A spatio-temporal model based on GCM and AGWL

This section presents 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.

4.1 A strategy for reuse and merging GCM and AGWL concepts

Figure 4 sums up our strategy to reuse GCM and AGWL principal concepts in order to define a spatio-temporal model. Let briefly discuss each choice before presenting them in more detail.

First, as a task-component is primarily a component, it makes sense to extend GCM component definition with a task definition. Moreover, contrary to an AGWL activity, a GCM component can be extended thanks to inheritance. It provides a powerful mechanism to improve code reuse. Second, as task-components are based on GCM, the port model is based on the GCM one. Hence, it should be extended to support temporal ports. With respect to AGWL, data input and output ports seems to be a satisfactory solution to represent temporal ports. Third, to describe an application, our proposal aims to meet the level of expressiveness offered by a workflow language, in our case the level of AGWL. Hence, our approach is to start from this language and to extend it with missing concepts. That mainly consists in replacing activities with components and in introducing spatial ports and their connections. Fourth, such an extended AGWL will drive the life-cycle of task-components:

```
interface ExtendedTypeFactory: TypeFactory{

 // inherits InterfaceType createFcItfType(...)
 // and ComponentType createFcType (InterfaceType[] itfTypes)

 // temporal port type definition
 TemporalPortType createFcTmpType(in string name, in string dataType,
                                  in boolean isInput, boolean isOptional,
                                  ...)
                  raises(InstantiationException);

 // a component's type definition with temporal ports
 ComponentType createFcType( in InterfaceType[] interfaceTypes,
                             in TmpPortType[] tmpPortTypes)
                  raises(InstantiationException);
};
```

Figure 5: Extending GCM with input and output ports types.

the control flow will mainly determine it but with additional constraints with respect to spatial compositions.

The remainder of this section reviews these points in more details.

4.2 Extending GCM components with tasks and temporal ports

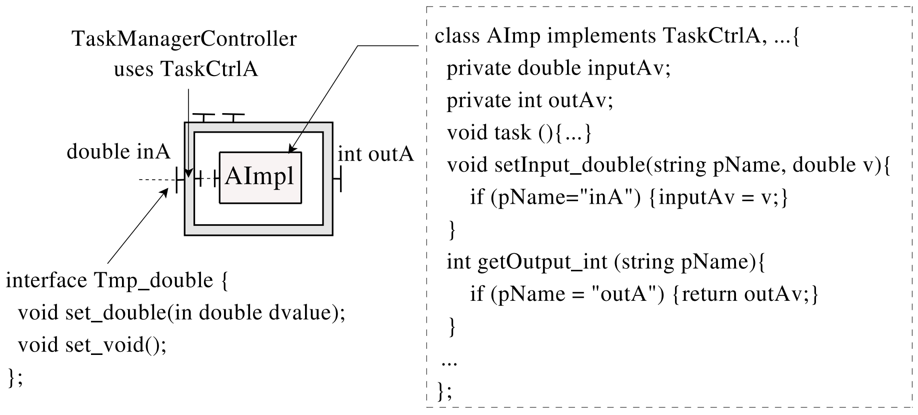
Extending a GCM component with temporal ports and tasks can be done in several ways. The objective of this section is to present the main requirements as well as a solution for supporting those concepts.

A component being defined by its ports, a new family of ports is needed to define a task-component. Let 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. Existing workflow languages support many data types such as primitive types (int, string, etc), files, packages, etc. As GCM defines typed interfaces, our model follows the same logic, but on data. Figure 5 illustrates an extension to the GCM *TypeFactory* interface dedicated to create types. The *createFcTmpType* operation creates the definition of an input (*isInput = true*) or output (*isInput = false*) port named *name* and for which the type is determined by the *dataType* argument. As temporal ports are distinguished from classical ones, a component type declaration is also extended to include this new kind of ports. To avoid any confusion, *InterfaceType*, *ComponentType* and *TemporalPortType* represent interfaces for retrieving information on a type.

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. How this operation is defined 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 and with no loss of generality, the support of only one task per component is presented in this paper. Once the inputs of a task are received, the task can

```
interface TaskController{
 void task();
};

interface TaskCtrlA: TaskController{
 void setInput_double(in string pName,
      in double v, in boolean isVoid);
 int getOutput_int(in string pName)
      raises NoValueException;
};
```



```
interface Tmp_double {
  void set_double(in double dvalue);
  void set_void();
};
```

```
class AImp implements TaskCtrlA, ...{
  private double inputAv;
  private int outAv;
  void task (){...}
  void setInput_double(string pName, double v){
     if (pName="inA") {inputAv = v;}
  }
  int getOutput_int (string pName){
     if (pName = "outA") {return outAv;}
  }
 ...
};
```

Figure 6: Interfaces associated to task and temporal ports and their usage.

be launched and once it finishes, output data should be sent to connected input ports. However, that should be the role of a framework. Hence, a task can be perceived from two point of view: a developer and a framework point of view. In the case of GCM, the framework role can be assigned to a dedicated controller (the stubs of the component). Hence, we extended the GCM specification with the interface *TaskController* described in the left of Figure 6. It represents the interface to be implemented by a developer to actually implement a task. Such an interface is accessed by a component controller, as viewed in the right of Figure 6.

Last, the usage of a temporal port has to be specified. The main requirement is to define how a task retrieves and sets its input and output data. It should be done from **1)** the implementation point of view for data usage and, from **2)** the outside for data transfer. To satisfy this requirements, our approach is to project the concept of temporal ports to a classical set of well defined client and server interfaces. The *TaskCtrlA* and the *Tmp_ double* interfaces presented in Figure 6 are examples for temporal port usage. They are associated to the c temporal ports of Component `A`. Let us briefly explain how they works:

Developer point of view To be able to get input data and set output ones inside a component implementation, the principle of attributes is chosen. That means, for each input data, when it is received on its corresponding port, a setter operation is called on the component's implementation, which is responsible to handle the data. Symmetrically, retrieving an output data from a component is done through a getter operation. The *TaskCtrlA* shown in Figure 6 represents setter and getter operations to be implemented by the component. As ports are named in GCM, a getter or a setter operation prototype relies on the temporal port types (*setInput_ DataType* and *getOutput_ DataType*) and names (*pName*) specified in a component type definition. Additional parameter *isVoid* or exceptions are also defined for particular cases, such as when there is no input data received or no output data produced. As temporal ports are strongly correlated to the task concept, corresponding operations are chosen to be defined in an extended *TaskController* interface. The way and the moment at which they are called are explained later.

External point of view Let explain the proposed external view of temporal ports starting from the input side. An input data is simply a data to be set once it is available and if the control reaches the task. Therefore, it seems to be natural to associate an external setter operation to a corresponding input port. That is the role of the interface *Tmp_ double* described in Figure 6. An input port may be then projected to a classical server one. However, our proposal allows the realization of effective connections between temporal ports. That should allow direct data transfer between components. An underlying execution environment (workflow-like engine) may adopt different scheduling, components instantiation and data transfer policies. For example, to instantiate a component `B` for which the task follows the one of a component `A` and consumes its output data, the execution environment can decide to instantiate `B`, connect its input ports to output ports of `A` and transfer data before removing `A`. According to such an alternative and to ensure the compatibility of connected ports, the type of an output port is projected to a client interface of the same type as of an input port (*Tmp_ double* in the example). Another scenario is however possible. The environment or the user may retrieve itself output data and set their values on input ports. The fact that a client interface is associated to an output port is not a limitation. The output data can be retrieved thanks to the *TaskCtrlA* interface which is a typical GCM server interface.

The requirements to support the temporal dimension within GCM are completed by tasks and temporal ports management concerns. From the proposed approach, it is expected to make transparent input data availability detection and a task execution triggering. It is also aimed to be able to easily detect the end of a task execution in order to subsequently acts for output data transfer and the component's life cycle management. Our approach proposes to extend the controller family of a GCM component with a particular controller named *TaskManagerController*. This paper does not present in details the specification of such a controller. Its role however can be sum up to: *1)* implementing the external interfaces associated to input ports, *2)* setting the user attributes associated to input data through the use of the *TaskController* interface, *3)* triggering the task execution and waiting its end, *4)* retrieving output data for direct transfer on connected output ports if needed, and *5)* maintaining a progress state of the task execution accessible from the outside of the component (through dedicated interface), especially for making decisions by the manager of the component life cycle.

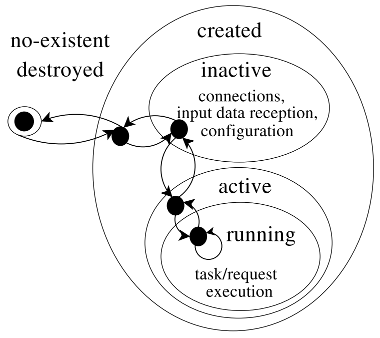


Figure 7: State machine diagram of task-components.

4.3 Life cycle management of task-components

Figure 7 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.

4.4 A composition language based on a modified AGWL

The application composition model proposed in this paper 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. In other words, we need to make AGWL GCM components aware. This section gives an overview of the impact of such a replacement according to the definition and composition of components. The term component refers to the GCM based task-component specified in the previous section.

```
<!-- AGWL activity -->
<activity name="name" type="type">
  <dataIn name="name"/>*
  <dataOut name="name"/>*
</activity>

<!-- AGWL sub-workflow -->
<subworkflow name="name">
  <dataIn name="name"/>*
  <body> <activity>+ </body>
  <dataOut name="name"/>*
</subworkflow>
```

```
<!-- modified AGWL : component type part -->
<component name="name" (extends="parentType")?>
 <dataIn     name="name" type="dataType"/>*
 <dataOut    name="name" type="dataType"/>*
 <clientPort name="name" type="interfaceName"/>*
 <serverPort name="name" type="interfaceName"/>*
 <attribute  name="name" type="attributeType"/>*
 <!-- other spatial port types -->
 ( <impl type="exe|dll|.." signature="sign" />
 | <body> <component>+ </body> )
 <controllerDesc desc="desc"/>?
</component>
```

Figure 8: From an AGWL activity and sub-workflow to a GCM task-component.

Component definition The type of a component is essentially defined by its ports (Figure 5). The internal structure of a component (binary code for a primitive component and composition description for a composite) represents

```
<component name="name" (extends="...")?>
 <dataIn name="name" (type="...")?
         set="outputRefORvalue"/>*
 <clientPort name="name" (type="...")?
             set="serverRef"/>*
 ...
 <attribute name="name" (type="...")?
            set="attrRef">*
 <body>
   ( <component>+
     <setPort client="name"
              server="nameOrData"/>?
   )*
 </body>
</component>
```

Figure 9: Main spatial composition elements.

```
<sequence name="name">
 <dataIn name="name" type="..." (set=..)?/>*
 <dataOut name="name" type="..."/>*
 <clientPort name="name" type="..." (set=..)?/>*
 <serverPort name="name" type="..."/>*
 <!-- other spatial ports -->
 <component>+
</sequence>
<if name="name">
 <!--  ports like in sequence-->
 <component>*
 <condition> condition </condition>
 <then> <component>+ </then>
 <else> <component>+ </else>?
</if>
```

Figure 10: Control structure examples.

configurable parameters to be applied when a component is instantiated. A same component type can be configured to be primitive or composite and with different implementations. The two kinds of components are then viewed as a sole concept when they are defined. In contrast, an atomic activity type in AGWL is distinguished from a sub-workflow, as described on the left of Figure 8. This distinction does not seem to be a necessity, as an internal composition description can be viewed as a particular implementation. Therefore, our proposal keeps the GCM logic and replace both activity and sub-workflow constructs by a sole component concept. The result is shown in the right of Figure 8. A component definition may inherit from existing ones, and may specify temporal, spatial or attribute ports (from line 3 to line 8)). It can also configure its internal structure thanks to the *impl* (line 9) or *body* (line 10) elements for respectively a primitive and a composite components, as well as its membrane description (*controllerDesc*, line 11). The content of a *body* element is explained hereinafter.

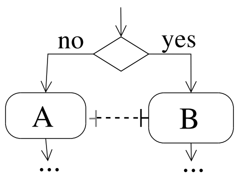
Composition The next step is to determine the impact of a spatio-temporal composition on the composition principles of the AGWL language. In particular, we focus on port connection/configuration, data flow composition, control flow composition and spatio-temporal composition.

Connecting and configuring ports A connection consists in configuring the value of a client or an input data port. The value of the configured port is respectively a reference to a server port or to an output data port. Figure 9 shows the enriched elements of a component definition for connection concerns. Two possibilities are offered to the user to configure a port. A port can be configured when it is defined, thanks to the *set* attributes (lines 3 and 5). Ports can also be configured latter. That is done thanks to the *setPort* instruction (lines 11,12) which is used to connect spatial and temporal ports. An attribute port (line 7) is a particular client port which does not require a connection. From the point of view of AGWL, the connection logic is almost the same. The sole difference is the addition of spatial ports connections.

Data flow composition As connecting an input data port to an output one is a direct mapping of a data dependency specification between tasks, data-flow composition remains the same as in AGWL.

Control flow composition In order to compose an application according to a control flow composition, our approach is to respect the composition principle offered by AGWL. That means, a control construct (sequence, if, while, etc.) is considered as a special kind of components. The particularity is that the internal structure of these components is pre-defined as well as an associated data-flow model for validity checking. A control structure can nevertheless define classical ports and can be reused as a component. Figure 10 presents two AGWL control structures adapted to our proposal. Compared to the component definition shown in Figure 9, the *body* of the sequence component is implicit. It is determined by all its internal components: the sequence is implied by the order of declaration. The two bodies of the *if* construct are however explicitly delimited by the *then* and *else* elements. Line 11 allows pre-declaring components for which the objective is explained below. As control structures appear as components, they can be reused everywhere within a component body.

Spatio-temporal composition With respect to the component specification shown in Figure 9, spatial, temporal and spatio-temporal compositions can be described. In addition, it is possible to connect ports, in particular spatial ports, at different levels of a composition. However, when components are implicated in a spatio-temporal composition, it may be required to determine the expected behavior. For example, let consider the composition presented graphically in Figure 11. Intuitively, the composition seems to be illegal with respect to only the temporal dimension. However, though the temporal dimension is aimed to mainly drive components life-cycle, the spatial dimension may also be taken into account. That should be possible, as a task is assumed to be an operation like other provided ones. Therefore, the fact that only one branch, for example the one of component `A`, is executed should not prevent the instantiation of `B`. To overcome such a confusion and to enable a composition to reflect as much as possible the suited behavior, we propose to define 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. Therefore, if two spatially connected components are implicated in the temporal composition of an application and if only one task belonging to*



```
<if name="name">                           <if name="name">
 ...                                        <comp. name="B" ...>...</>
 <then> <comp. name="B" ...>                <comp. name="A" ...>
        <svrPort name="pB" .../>             <cltPort ...  set="A.pA"/>
       </comp.>                             </comp.>
 </then> <else>                             ...
  <comp. name="A" ...>                      <then> <comp. refName="B"/> </then>
   <cltPort ... set="A.pA" />               <else> <comp. refName="A"/> </else>
  </comp.>                                 </if>
 </else> </if>
```

Figure 11: Illegal (left) and legal (right) compositions because temporal or spatial dimension prevails.

one of two components is reachable by the control flow then, in the first case, the composition is considered illegal. Thus, the graphical composition of Figure 11 is valid if the spatial connection is defined outside the two bodies of the *if* construct (outside the construct or in the pre-declaration space), as illustrated on the right of Figure 11.

5 Example of an application description

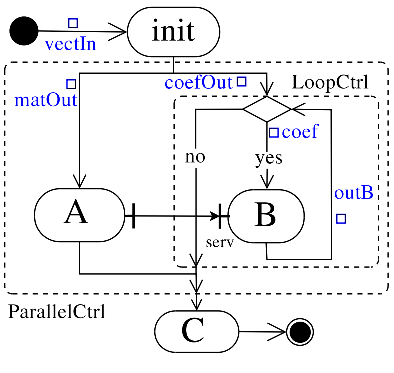


Figure 12: Application example.

Figure 12 shows a simplified application composed according to the proposed spatio-temporal model. 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 the component `init`. The results computed by `A` depends on parameters provided by the component `B`. These parameters vary in the time depending on the iterative `B` computation realized thanks to the loop control construct. It is expected that the component `B` has a persistent active state for continues `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 13 illustrates how the application described herein before can be modeled within the proposed modified AGWL language. The expressed execution ordering matched perfectly with the specified requirements. First, the simultaneity of the execution of the `A` and the `B` component tasks is ensured thanks

```
1  <assembly name ="example">
2   <dataIn name="vectIn"  type="Vect"/>
3   <body>
4    <sequence name="Seq">
5     <component name="init">
6      <dataIn name="vectIn" type="Vect" set="vectIn"/>
7      <!-- out: Matrix: matOut; Double: coefOut -->
8     </component>
9     <parallel name="ParallelCtrl">
10     <component name="B">
11      <!-- in: Double: inB, out: Double: outB -->
12      <serverPort name="pB" type="GetRes"/>
13     </component>
14     <component name="A">
15      <dataIn name="inA" ... set="init.matOut"/>
16      <clientPort name="pA" type="GetRes" set="B.pB"/>
17     </component>
18     <section>
19      <component refName="A"/>
20     </section>
21     <section>
22      <while name="LoopCtrl">
23       <dataIn name="coef" type="Double"
24                           set="init.coefOut"
25                           loopSet="B.outB"/>
26       <condition> coef < 1000 </condition>
27       <loopBody>
28        <component refName="B">
29         <dataIn name="inB" set="coef"/>
30        </component>
31       </loopBody>
32      </while>
33     </section>
34    </parallel>
35   </sequence>
36  </body>
37 </assembly>
```

Figure 13: Application description in modified AGWL.

to their integration in *parallel section* elements (lines 18 to 33) of the *parallel* control structure (lines 9 to 34). Second, the spatial connection between `A` and `B` (line 16) outside the loop body (lines 27 to 31) explicits the possibility of having a persistent active state for `B`. In addition, according to the specified rules to determine components life-cycle, this state remains valid (even after the end of the loop execution) until the component `A` reaches the inactive state, in other words, the end of its internal task(s). As a result, the proposed spatio-temporal model was able to express a coherent behavior with a well defined life-cycle of a components. It should be enable efficient automatic scheduling decisions.

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 strong 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.

After analyzing what composition in space and in time means, the paper has analyzed some possible designs of models combining both of them. As a result, the paper describes such a model based on two existing models – GCM as a component model and ASKALON as a workflow model – so as to leverage existing models. We made the decision to extend GCM with temporal ports and task concepts and to adapt AGWL to offer a spatio-temporal composition language. Some benefits of the approach has been illustrating with respect to a motivating application.

As this paper mainly presented a model, the next step is to actually implement it. The implementation of the component part of the specification appears quite straightforward. A major piece of work concerns the support of the modified version of AGWL. Therefore, we will study whether it is more relevant to adapt the AGWL workflow engine, to re-implement it or maybe to map the modified AGWL to plain AGWL. Though the latter should not lead to an efficient implementation, it may be enough to validate the proposed model as well as to study more in depth the state machine transitions: as several spatio-temporal semantics appear meaningful, we need to understand what it is the more adequate semantic for application developers.

References

[1] I. Altintas, A. Birnbaum, K. K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, and Yohann. A framework for the design and reuse of grid workflows. In *First International Workshop on Scientific Applications of Grid Computing (SAG'04))*, pages 120–133, Berlin/Heidelberg, 2005. Springer.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. Technical report, May 2003.

[3] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[4] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 40–48, Washington, DC, USA, August 2006.

[5] E. Bruneton and T. Coupaye and J.B. Stefani. The Fractal Component Model, version 2.0-3. Technical report, ObjectWeb consortium,, Feb. 2004.

[6] T. Fahringer, J. Qin, and S. Hainzer. 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, pages 676–685, Cardiff, UK, May 2005.

[7] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of component applications within a grid environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[8] P. M. Institute. Basic features of the grid component model. Technical report, CoreGRID, Mar. 2007. D.PM.04.

[9] M. Beisiegel and H. Blohm and D. Booz and M. Edwards and O. Hurley and S. Ielceanu and A. Miller and A. Karmarkar and A. Malhotra and J. Marino and M. Nally and E. Newcomer and S. Patil and G. Pavlik and M. Raepple and M. Rowley and K. Tam and S. Vorthmann and P. Walker and L. Waterman. SCA Service Component Architecture - Assembly Model Specification, version 1.0. Technical report, Open Service Oriented Architecture collaboration (OSOA), Mar. 2007.

[10] OMG. CORBA component model, v4.0. Document formal/2006-04-01, Apr. 2006.

[11] OMG. Unified modeling language: Superstructure, version 2.1.1. Document formal/2007-02-05, Feb. 2007.

[12] S. Pllana and T. Fahringer. Uml based modeling of performance oriented parallel and distributed applications. In E. Yucesan, C.-H. Chen, J. Snowdon, and J. Charnes, editors, *In proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.

[13] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[14] F. Thomas, P. Radu, D. Rubing, N. Francesco, P. Stefan, Q. Jun, S. Mumtaz, T. Hong-Linh, V. Alex, and W. Marek. ASKALON: A Grid Application Development and Computing Environment. In *Proceedings of the 6th International Workshop on Grid Computing*, pages 122–131, Seattle, USA, November 2005.

[15] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, september 2005.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399