Towards Software Component Assembly Language Enhanced with Workflows and Skeletons

Marco Aldinucci
Dept. of Computer Science - University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy
aldinuc@di.unipi.it

Hinde Lilia Bouziane
INRIA/IRISA, Campus de Beaulieu
35042 Rennes cedex, France
Hinde.Bouziane@inria.fr

Marco Danelutto
Dept. of Computer Science - University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy
marcod@di.unipi.it

Christian Pérez
INRIA/IRISA, Campus de Beaulieu
35042 Rennes cedex, France
Christian.Perez@inria.fr

ABSTRACT

We explore the possibilities offered by a programming model supporting components, workflows and skeletons. In particular we describe how STCM (Spatio-Temporal Component Model), an already existing programming model supporting components and workflows, can be extended to also provide algorithmic skeleton concepts. Programmers are therefore enabled to assembly applications specifying both temporal and spatial relations among components and instantiating predefined skeleton composite components to implement all those application parts that can be easily modeled with the available skeletons. We discuss preliminary results as well as the benefits deriving from STKM (Spatio-Temporal sKeleton Model) adoption in a couple of real applications.

Keywords

Software Component, Skeleton, Workflow, Abstraction, Grid.

1. INTRODUCTION

Grids as well as recent large scale parallel machines propose a huge amount of computational power and storage. Therefore, it is possible to envision scientific code coupling applications that solve problems related to bigger or different, not yet solved physical phenomena. A major issue still to be solved is the design of a programming model suitable to ease application development and to efficiently exploit resources.

Let us consider some of the important properties that have to be provided by such a programming model. A first property is to face the complexity of software management, and in particular to enable code reuse. Second, it should support strong coupling algorithms that are often present in high performance applications. Third, as resources are more and more shared, the programming model should enable an efficient usage of resources, in particular through the support of loosely coupled application elements. Fourth, it should abstract resources to achieve two important goals: let the programmers to only deal with functional concerns – non functional concerns must be hidden – and applications should be portable to a wide range of architectures – provide abstractions that can be adapted to resources.

There are many programming models that attempt to ease programming large complex scientific applications and to hide the complexity of underlying execution resources especially Grid infrastructures. This paper focuses on those based on assembly/composition principle, as programming by assembly is gaining increasing acceptance to deal with complex scientific applications. In particular, it deals with three well-known model families: software component models, workflow languages and skeleton based programming models. Each family attempts to tackle with the presented properties to deal with the complexity of applications and/or resources. Depending on a given model, the properties are less or more handled. For example, modern software engineering practices promote the usage of software component models [29] to deal with code reuse. In particular components enable to easily build an application made of piece of codes written in different languages. While component models appear adequate for strong coupling composition, workflow models seems more tailored for loosely coupled compositions. In addition, algorithmic skeletons are considered better suited to provide a simple abstraction that can be automatically optimized to the resource of the system [17]. Hence, there is no model that efficiently handles all these properties. Though all these properties are relevant, they should be all and well considered by a single programming model. As far as we know, there is not such a model. Nonetheless, there are some previous works aimed at bringing closer these families. For example, STCM (Spatio-Temporal Component Model) [14] is a model combining component models and workflows. Similar efforts have been carried out for skeletons and component models [3, 20].

This paper explores the feasibility of a programming model combining the three families – components, workflows and skeletons. Rather than proposing a programming model from scratch, it studies how to combine STCM– which already unifies components and workflows – with skeletons. The outcome should be a programming model supporting all the presented properties.

The remainder of this paper is organized as follows. Sections 2 and 3 recap main features and technical background of component-and-workflow and skeleton-and-component methodologies, respectively; STCM and *behavioural skeletons* are presented as

CBHPC 2008, October 14–17, 2008, Karlsruhe, Germany.

paradigmatic examples of the two methodologies, which are compared in Section 4. Section 5 introduces STKM (Spatio-Temporal sKeleton Model), where the two methodologies are stacked in a two-tier architecture aiming at raising the level of abstraction of both component-based and workflow-based parallel/distributed programming approaches. The benefits of the approach are shown by reasoning about the design of two real-world applications (biometric identification and climatology applications). Section 7 concludes the paper and presents future works.

2. STCM: MERGING COMPONENT MODELS AND WORKFLOW LANGUAGES

In [14], we proposed a Spatio-Temporal Component Model (STCM). This model combines two technologies: software component models and workflow models. Its aim is to allow a designer to express the behaviour of an application by assembly. This behaviour considers both the temporal logic of the application execution, based on reusing workflow concept, and the spatial dependencies that may exist between components, based on reusing component assembly concept.

Before giving an overview of STCM [14], let us introduce software component models and workflow languages. The introduction is done according to a generic view of existing technologies and to the main properties that motivate the combination of the two approaches in STCM.

2.1 Software Component Models

Independently of existing technologies, like CCA [11], CCM [26], GCM [18] or SCA [9], a software component appears as a black box unit of a reusable, composable and deployable code. The composition is done through the connection of well-defined ports that allow a component to interact with other components. The interaction between two components often follows a *provide-use* paradigm. According to existing component models, this paradigm is mainly based on one of the following communication models: operation/method calls, message passing, document passing (Web Services), events or streams. Most of the existing assembly models for components exploit bindings based on spatial relationships. That means that the bound components are concurrently active for the entire period they are bound; the frequency of the interaction between components is usually not known. As result, an application assembly corresponds to its architecture at execution. These kind of architectures are captured by UML component diagrams [27].

2.2 Workflow Languages

Many environments exist [33] that offer workflow based programming models to develop and execute scientific applications. Examples are Askalon [22], Triana [30], Kepler [6] and BPEL [7]. In general, building an application according to a workflow means describing the order of actions, often named *tasks*, which should be executed and their data dependencies. For that, control flow and data flow models are proposed. A control flow model allows the description of the execution order of tasks by using control constructs such as sequences, branches or loops. A data flow model focuses on data dependencies between tasks. To define data dependencies, a task specifies its inputs and outputs ports. Thus, describing connections of output ports of some tasks t_i to input ones of a task T defines data dependencies between t_i and T. Therefore, workflow models deal with temporal compositions. These kinds of compositions are also captured by UML activity diagrams [27].

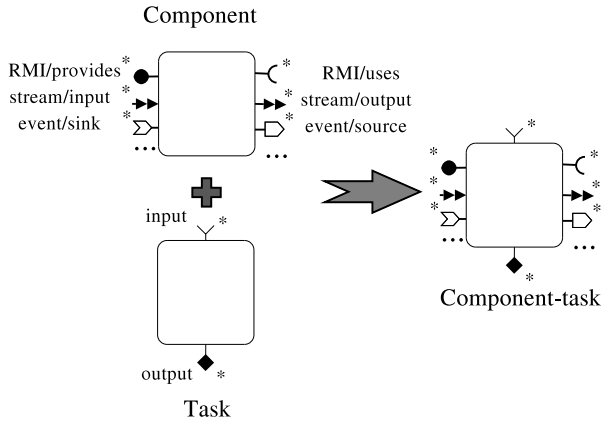


Figure 1: A component-task as a combination of a component with task concepts.

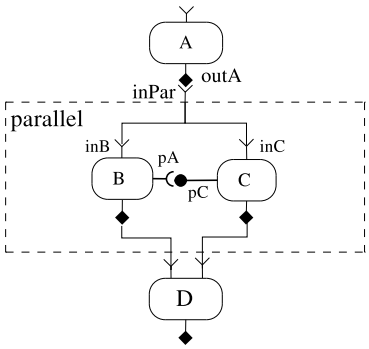
2.3 STCM

Component models offer well-founded concepts for code reuse and applications complexity management. However, while the spatial property of component models make them more appropriate to develop strong coupled applications, the temporal property of workflow models eases the programming of the temporal logic of an application that can be moreover captured from the assembly to enable efficient resources management. In order to group the advantages of the two programming approaches, STCM proposes a combination of component models and workflow languages. For that, it defines the concept of *component-task*, a spatio-temporal assembly model and life cycle management. Let us give an overview of these concepts.

Component-task. As shown in Figure 1, a component-task is a component that supports the concept of task. Thus, in addition to classical ports, named *spatial* ports in STCM, a component-task can define input and output ports, named *temporal* ports. Temporal ports and task behave like in workflow models. The difference is that the life-cycle of a component-task may be longer than the one of a task in a workflow, which usually corresponds to its execution. In addition, a task in STCM can communicate with other component-tasks through client spatial ports. More details about the specification of task and temporal ports concepts can be found in [14]. This specification is presented through an extension of a GCM (Grid Component Model) component.

Spatio-temporal assembly model. The assembly model proposed for STCM is inspired from the *Abstract Grid Workflow Language* [22] (AGWL)[1]. AGWL offers a hierarchical model made of atomic and composite tasks. A composition is done with respect to both data flow and control flow compositions. The control flow supports several control constructs like sequences, branches (*if* and *switch*), loops (*for* and *while*) and parallel constructs (*parallelFor* and *parallelForEach*), etc. The assembly model of STCM is mainly based on replacing an AGWL task by a component-task, including the addition of spatial composition.

[1]Other workflow languages can be chosen. The principle of modifying them to define a spatio-temporal assembly model is similar.




```
1  component Example {
2    ...
3    parallel parCtrl {
4     dataIn Double inPar <= a.outA;
5     // declarations
6     component B { dataIn Double inB;
7                   clientPort Compute pB;
8     };
9     component C { dataIn Double inC;
10                  serverPort Compute pC;
11    };
12    instance B b;
13    instance C c;
14    connect b.inB  to parCtrl.inPar;
15    connect c.inC to parCtrl.inPar;
16    connect b.pB to c.pC;
17
18    // instructions
19    section : exectask (b);
20    section : exectask (c);
21  } // end parallel
22   ...
23 }
```

Figure 2: Simplified example of an assembly recalling the principle of STCM.

Figure 2 gives an example of a composition using a simplified STCM assembly language (the original syntax is in *XML* format for which a grammar is presented in [14]). The textual part shows the assembly inside the `parallel` control structure. The proposed language allows to declare component-task types (`B`, lines 6 to 8, and `C`, lines 9 to 11) and component-task instances (`b`, line 12, and `c`, line 13), describe a data flow (lines 4, 14 and 15), describe the order of execution of tasks (`b` and `c` in parallel, lines 19, 20) and spatial dependencies (line 16). Means are then offered to a designer to build by assembly an algorithmic logic, including temporal and spatial dependencies at the same level of a composition. That is relevant not only to simplify the design but also to increase code reuse and be able to envisage solutions for efficient usage of execution resources.

Life cycle of a component-task. To manage the life cycle of component-tasks during an application execution, STCM defines a dedicated model. The management relies on the ability of capturing the algorithmic logic directly from the assembly and ensure for example safe destruction of component-task instances. For that, STCM defines a state machine diagram corresponding to the life cycle of a single component-task and an assembly semantic to reflect as much as possible a deterministic application behaviour.

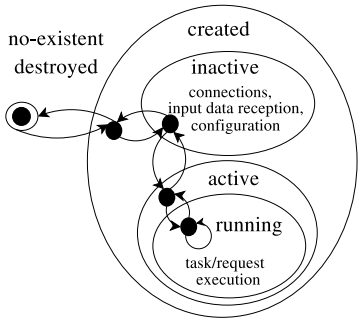


Figure 3: Life cycle of a component-task.

The state machine is recalled in Figure 3. From this diagram, it can be observed that the activation duration of a component-task instance can be longer than a task execution duration. Also, a component-task can be active without any running task. This is required when a component-task provides functionality on which other component-tasks depend in the assembly. For instance, in the assembly shown in Figure 2, component `C` can be activated by the arrival of an input data on `inC` port or when an operation is invoked on `pC`. Then, the activation duration of `C` doesn't depend only on the execution duration of `C's` task but depends also on the activation duration of component `B` which uses `C`.

The semantic associated to an STCM assembly is determined with respect to simple composition rules to be taken into account when building an application. The main rules are the following:

- *If a component-task A uses (composition in space) a functionality provided by another component-task B, then B must be concurrently active with A and remains active as long as A is active.*
- *If a component-task instance A or if another component-task B that uses A is no more reachable by a control flow, then A becomes useless and can be destroyed.*
- *A component-task instance must be activated at the latest when the control flow reaches the execution of its task and input data are received or when it is used by another component-task instance.*
- *The execution of a task is assumed to produce not more than one output data on a same output port.*

Besides the proposed assembly constructs, these rules are expected to help a designer to easily express a suited behaviour. They aim also to ease automatic management of an application structure with efficient resources usage.

3. SKELETON BASED PROGRAMMING

Structured parallel programming models based on the algorithmic skeleton concept are around since the '90s since skeleton concept introduction by Cole [16]. Later on, several research groups developed programming environments, systems and libraries based on the skeleton concept [8, 21, 25, 10, 23, 31]. Skeleton based programming models allow programmers to express parallelism using a set of predefined patterns, the skeletons, that model common parallelism exploitation patterns. Typical skeletons are either stream

or data parallel. Classical stream parallel skeletons are pipelines (modeling computations performed in stages) and farms (embarrassingly parallel computations). They exploit parallelism between computations of different input tasks of the input stream to produce a stream of results. Typical data parallel skeletons are map (independent forall), reduce (summing up of a collection of data via an associative and commutative operator) and stencil (forall with dependencies). They all exploit parallelism in the computation of a single input task.

The skeletons are parametric and programmers can therefore customize them by defining the kind of primitive computation used by the skeleton (e.g. a pipeline stage or a farm/map worker), its parallelism degree or any other kind of skeleton specific features (e.g. whether or not a farm should guarantee input/output ordering). Most likely, skeleton programming environments and systems allow programmers to nest skeletons (e.g. a pipeline stage can be expressed as a farm/map skeleton) and therefore skeleton based applications happen to be *structured* as a skeleton nesting *plus* some sequential code used as a parameter for the leaf skeletons.

Once applications have been structured via proper skeleton nesting, the implementation of the skeleton framework takes care of all the aspects relative to parallelism exploitation. Parallel activities setup, mapping and scheduling, communication and synchronization handling and performance tuning are all aspects that are dealt with at the skeleton implementation level rather that in the programmer application code. Being the skeletons known and efficient patterns of parallelism exploitation, this results in very efficient and scalable application implementation, independently of the model chosen for the implementation, that traditionally is either template based [28] or macro data flow [19]. Overall the whole process results in a complete and worth *separation of concerns* between application programmers and system programmers. The former are in charge of recognizing parallelism exploitation patterns in the application at hand and of modeling them with suitable skeletons (or skeleton nesting). The latter are in charge of solving, once and for all, when the skeleton framework is designed and implemented, the problems related to the efficient implementation of the different parallelism exploitation patterns and to their efficient composition. This separation of concerns has a notable list of positive side effects: i) it consistently contributes in supporting rapid application development and tuning, ii) applications programmers are not required specific knowledge on parallelism exploitation techniques, iii) programs can be seamlessly ported to different architectures provided that system programmers have already studied, designed and implemented proper skeleton implementation for the new target, just to mention a few.

Algorithmic skeletons can be quite easily associated to software components. A skeleton is a building block for parallel applications exactly the same way a component is a building block for a generic application. As a consequence, skeleton technology has recently been used in the component based programming scenario [2, 23]. In this case, (composite) components are provided to the user that model common parallelism exploitation patterns and accept other components as parameters modeling the skeleton inner computations (e.g. the pipeline stages or the farm workers).

The last step we want to mention here in the algorithmic skeleton concept evolution has been the introduction of autonomic management aspects in skeletons. Skeleton implementation was in charge of handling all the non-functional aspect of parallelism exploitation since the very beginning. However, the advent of significantly new architectures, such as grids, with highly dynamic and unreliable features imposed some more evolved approach to non-functional aspect handling. Therefore, autonomic management of skeleton features has been introduced [4, 3] that dynamically adapts skeleton execution to the varying features of the target architecture considered. Using this "last" version of the skeletons (named *behavioural skeletons*, to explicitly mention they have managers taking care of dynamic behaviour of the skeleton implementation) users can develop (grid) applications that seamlessly and *without any kind of user/application programmer intervention* react to node faults, additional node loads, network inefficiencies and keep (in a "best effort" way) the application running according to a user specified QoS contract.

4. STCM *VS.* SKELETONS: DISCUSSION

Despite the ability of STCM to abstract the behaviour of an application through its assembly, the level of abstraction remains low. This is the case in particular for parallel programming. In this context, two issues must be taken into account. This section introduces and discusses these issues and motivate the work presented in this paper.

The first issue is related to the design of parallel programming paradigms using STCM. The relations that can be expressed between component-tasks in STCM remain simple. In the spatial dimension, only relations of type 1-to-1 or 1-to-N can be expressed between assemblies of component-tasks. While in the temporal dimension, only simple tasks and data parallelism can be expressed through control constructs like `parallel` or `parallelForEach` (independent forAll). Even if a combination of the two can reach more complex behaviour, offered constructs are not sufficient to simply express a usage of complex parallel paradigms. This lead the designer to construct complex applications in arbitrary way and to consider parallelism issues when programming, thus increasing the likely of (inadvertently) introduce bottlenecks and/or execution resources dependencies in the design. As an attempt to overcome such a limitation, a first objective of the present work is to propose means to take benefits from skeleton principle to construct complex parallel applications in a simple way.

The second issue is related to efficient execution of an assembly. This issue relies essentially on scheduling policies adopted by an execution framework. A simple policy can consider the execution of an application step-by-step mainly directed by the temporal dependences between component-tasks. However, a more efficient scheduling should consider a global behaviour of part or whole application assembly, in particular to exploit maximum parallelism. For that, means are required to recognize parallelism forms from an assembly. Therefore, the second contribution of this paper aims to consider the extension of STCM with respect to resolving the first issue and analyze the possibility of exploiting parallelism behaviour from a component-task assembly. In this context, we propose to study the projection of an abstract assembly to skeleton based forms. We can then take benefits from already existing skeleton management mechanisms to efficiently execute an application.

5. TOWARDS STKM: A COMBINATION OF STCM WITH SKELETON BASED PROGRAMMING

In this paper we propose a combination of STCM and skeleton principles in the STKM model. The objective is twofold. The first goal is to increase the abstraction level of STCM regarding the programming of parallel applications. In particular, we aim to offer to a designer a programming approach based on skeleton constructs. That is to promote simplicity of programming, the construction of correct programs and code reuse. The second goal is to offer means for efficient execution of an application. For that, we propose to analyze the possibility to exploit parallelism behaviour from an assembly and follow a management approach based on a projection of the assembly to a composition of nested skeleton constructs. Thus, the management of parallelism can be turned to skeleton management for which a lot of efforts are already done to deal with low-level parallelism concerns and efficient execution.

This section presents our proposal in three parts. The first part presents the proposed extension of STCM regarding the support of skeleton constructs (Section 5.1). The second part outlines the consequence of defining STKM on top of STCM on STCM itself (Section 5.2). The last part presents the principle of managing the execution of an STKM application (Section 5.3).

5.1 Skeleton Constructs on top of STCM

Our approach to enable a designer to express the usage of skeleton-based parallel paradigms is to extend STCM with dedicated constructs. These constructs are particular composite components (templates) for which the internal structure is well defined according to a parametric schema. They can define ports and be composed with other skeleton constructs and/or components. The elements of a skeleton (stages for the pipeline and workers for the functional replication) can be skeletons or components (primitive or composite). These elements can also be composed with other components (internal or external to the skeleton construct). The objective is to promote composition at different levels, which should improve composability and code reuse, while preserving the pragmatics of skeletons. The extension of STCM consists in extending its assembly language [14]. An overview of this extension for the pipeline and functional replication skeletons is shown in Figure 4.

```
component  ::= stcmComp | skeleton
...
skeleton   ::= <skeleton name=string>
                  inputSkel? outputSkel? port*
                  attribute* skelConst?
               </skeleton>

inputSkel  ::= <inputSkel name=string  type=string
                                       (set=string)?/>
outputSkel ::= <outputSkel name=string type=string/>

skelConst  ::= pipe | funcRepl | sequential ...

pipe       ::= <pipe name=string>
                  inPipe
               </pipe>
inPipe     ::= component* instance* stage+
               configport *
stage      ::= <stage name=string>
                  skeleton
               </stage>

// Functional replication behavioural skeleton
funcRepl   ::= <funcRepl name=string>
                  inFuncRepl
               </funcRepl>
inFuncRepl ::= component* instance* worker
               configport* emitCollect? sharedComp?

      // emitcollect specifies the policy of
      // handling skeleton inputs and outputs
      // example (broadcast, reduce)
      // sharedComp specifies a component
      // encapsulating a shared state between workers

worker     ::= <worker name=string (cadinality=int)?>
                  skeleton
               </worker>

emitCollect::= <emitCollect emit=string
                            collect=string/>
sharedComp ::= <sharedCompInstance ref=string/>

sequential ::= stcmcomponent
...
configport ::= clientserv | inout
clientserv ::= <setPort client=string server=string/>
inout      ::= <setPort in=string out=string/>
```

Figure 4: Overview of the STKM grammar related to the skeleton composition part. Only pipe and farm constructs are considered. In bold, the grammar keywords. In italic, the STKM language keywords.

A skeleton in STKM defines at least its inputs/outputs (`inputSkel` and `outputSkel` in the grammar) and their functional elements. The input and output ports are not a new kind of ports. They are of stream type (as in classical skeleton usage) and are used to identify which component ports have the role of receiving and producing data proper to the skeleton computations. Therefore, a component can be reused by a simple wrapping mechanism (Figure 5). It is relevant to note that the wrapped component behaves like in a classical skeleton: a computation is started on the reception of an input data; the computation produces an output data on the output port. Otherwise, the behaviour of the skeleton is not preserved. In this regard, skeleton inputs and outputs can be bound to classical stream ports or temporal ports, in which case the computed function is a task. The latter case is a good example because it responds to suited behaviour. That is true thanks to the last STCM semantic rule defined in Section 2.3. For simplicity, in this paper, we assume that component-tasks define only one input and/or output port (if the task has data dependencies).

Figure 6 sketches an example of an STKM assembly. It illustrates the possibility of composing components with a skeleton construct and skeleton nesting. Compared with a classical usage of skeletons, it is easy in STKM to assemble sequential with parallel codes, when only part of an application is parallel. Moreover, a skeleton and its included components can define classical STCM ports and be composed with other components. This promotes expressing code dependencies by assembly rather than implementing them in the skeleton computation codes; that ease programming and improve code reuse. In addition, more complex behaviour can be expressed by a skeleton, like the possibility of accessing a shared state between computation codes in a functional replication skeleton (S component in Figure 6).

5.2 STCM modification requirements

STKM aims also to enable exploiting parallelism in several situations, in particular, in both spatial and temporal dimensions of an assembly. Even if the parallelism built by a skeleton construct infers a spatial assembly, which can be of course implicated in a temporal dimension (like shown in Figure 6), that may be not sufficient to ease expressing some behaviours. A typical situation is to express through an assembly that ordered tasks in part of a workflow

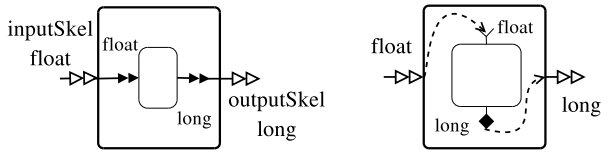


Figure 5: Wrapping a component to be a skeleton element. On the left, skeleton inputs and outputs are bound to stream ports. On the right, they are bound to temporal ports. The type of ports are data types which must be compatible.

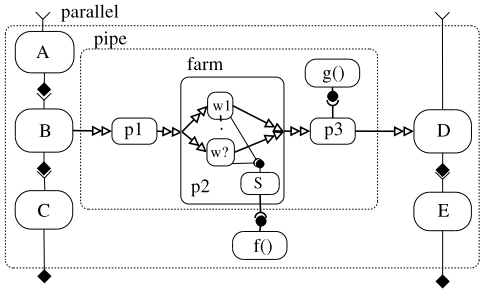


Figure 6: Example of a composition using STKM.

should be executed in a pipeline way. The left part of Figure 7 illustrates such a situation for a sequence. Syntactically, the proposed extension allows such a composition. However, the possibility of a pipelined execution depends on the ability of receiving multiple input data on the input stream of the pipe construct. As we assumed in STCM that not more than one output data on a temporal port may be produced for a single item and as the model preserves the semantic of control constructs, a mechanism is needed to be able to support such a situation. A mechanism is also needed to enable the collection of the results on a stream after a pipelined execution.

A solution is to relax the assumption specified in STCM to allow a task to produce multiple output data for a single input data and symmetrically, allow a task to collect multiple input data to produce one output data. For that, two issues are to be resolved.

First, it is necessary to enable a component-task to express the related task's behaviour when it is defined or composed. Otherwise, it may be difficult to determine the behaviour of an assembly. We propose to resolve this issue with a simple cardinality principle to be associated to temporal ports. The right part of Figure 7 shows the principle of the solution. An input port with cardinality 1 (respectively n) needs one data (multiple data) to execute a task. In the case of multiple data, the number of received data is determined by the end of the execution of the task that produces the data. An output port with cardinality 1 (respectively n) indicates that one data (multiple data) will be produced by one execution of a task.

The second issue is related to the need of a mechanism that allows a task implementation to be able to send (respectively receive) multiple data on output (resp. input) temporal ports. To produce multiple data, our solution consists in offering a callback operation to component-task implementation allowing a task to signal the availability of output data to be sent. This operation can be called multiple times. The end of the execution of the task corresponds to the end of producing output data for a single input data. The principle of this solution is already proposed in preliminary spatio-temporal composition model that we presented in [13]. Because a cardinality n for an output port affects the implementation of a component-task, the cardinality has to be specified in the definition of the port. On the input side, we assume that it is at the responsibility of the framework implementation to wait all incoming data before executing a task. In this case, the task behaves like in the case of having a single data received on the port. Therefore, it is sufficient to specify a cardinality n for an input port at the assembly level to obtain the suited behaviour. This, a component-task with an input port of cardinality n appears in an assembly as a reduction or synchronization point within an assembly.

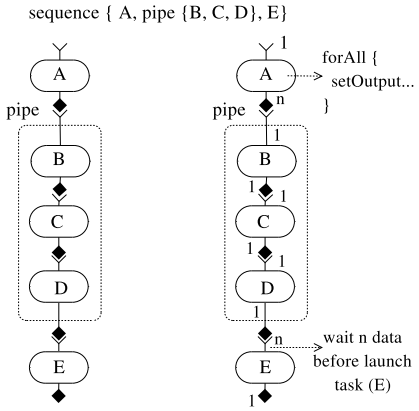


Figure 7: STCM modification to support skeleton constructs in temporal dimension: temporal ports cardinality principle.

The outlined changes in STCM raise the issue about their consequence on the life cycle of component-tasks and so on the semantic of an STKM assembly. The principle of a task is still dependent on the availability of one data. Even if it can produces multiple data, the end of its execution is still well determined. In addition, in STKM, the life cycle management is still directed by spatial and temporal dependencies between components, including skeleton constructs, for which the principle is the same as in STCM. The only modification affects the last semantic rule defined in Section 2.3 and which becomes: *"The execution of a task can produce multiple output data on a same output port. The end of the execution determines the end of producing all the output data."*. Finally, STKM preserves the global principle of STCM.

5.3 A suited approach for efficient execution management

Until now, we dealt with the abstract viewpoint of STKM offered to a designer. The goal of proposing such an abstraction is not limited to simplifying programming and improving the expressiveness of an assembly or improve code reuse, but also aims to make it possible to adapt an application to a given dynamic execution context while ensuring a given user-defined Quality of Service (QoS) contract. We showed in previous work that skeletons [3, 31, 4, 17] have the ability to cope with the autonomic steering of application execution to ensure dynamically defined levels QoS, and that it can be done while preserving their high-level nature ensuring good proprieties such as: the separation of concern between functional and management code (thus code reuse), the automatic generation of binary code (thus rapid prototyping and code portability), etc. In this regard, the approach has proved to be effective with respect to a number of domains, such as performance [4], security [5], and

fault tolerance [12][2].

Hence, an issue is to propose an approach to manage the execution of an `STKM` application. In general, the effectiveness of an execution depends on the expressiveness power of an assembly and the ability of an execution framework to *recognize* the behaviour of an application, to take into account execution resources (number of processors, size of memories, network architecture, availability and dynamicity of resources, etc.) and to make adequate decisions to adapt the application to the resources. Specifically, *behavioural* skeletons attack this problem (a.k.a. idiom recognition problem) by providing pre-defined parametric patterns exhibiting a well-defined behaviour, and thus, supporting pre-defined management strategies [3]. Thus, behavioural skeletons abstract component self-management in component-based design as design patterns abstract class design in classic OO development.

In the context of STKM, such decisions are expected to consider in addition to temporal and spatial dependencies, made by an STCM engine, the skeleton constructs. With respect to skeleton constructs, the main role of an STKM framework is expected to project or transform an STKM assembly to a concrete one (the assembly at execution). The projection consists in replacing a skeleton description in the abstract assembly by an adequate implementation. For that, our aim is to reuse already proposed component based implementations (such as behavioural skeletons in the GCM [3, 24]) and take benefits from their self adaptive management of computational elements and their ability to deal with optimization issues, like collapsing stages of pipes or introducing farms for efficiency. Following such an approach, an assembly after a skeleton construct replacement is expected to be an STCM assembly.

Since STKM skeleton deployment and activation is driven by temporal dependencies, they are dynamically deployed, and since they are parametric patterns, they can be dynamically configured at deployment time (e.g. according to available platforms). This kind of flexibility covers an additional case with respect to autonomic management (that is fully dynamic), compile-time configuration (static) and application launch-time malleability (launch-time) because each specific skeleton can be configured at the time it is really needed. This time may happen to be in a point of time well after the application launch, especially in very long running applications. This, in turn, may reflect in very different execution environments in the two points in time. We envision, as immediate result, the iterative mapping of the same skeleton (within a temporal loop) onto different reservations of grid sites along time. Observe that, for some kind of applications, flexibility may be as effective as fully dynamic adaptivity but, in general, it incurs quite lower adaptation overheads [4, 3].

In addition to the management of skeleton constructs, we are investigating the possibility of managing some parallelism forms that are not explicitly expressed by the usage of skeleton constructs but which can be mapped to a skeleton composition without modifying the expected behaviour. An example is to deal with the independent `forAll` control constructs (`parallelForEach`). The parallelism expressed by this construct can be mapped to a functional replication skeleton in which the workers are the body of the loop. Other parallelism forms can be also built in STKM purely based on the usage of temporal port cardinality principle. For example, if we assume that the pipeline construct shown in Figure 7

[2]those domains are all considered "in insulation" in these works, the multi-domain management is currently under investigation.

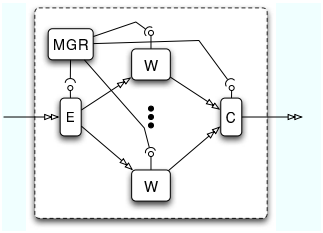


Figure 8: Functional replication behavioural skeleton component.

is not used and the cardinality on the ports are kept, an implicit pipeline behaviour is built. The ability of a framework to capture such a behaviour, which can be directly done thanks to the cardinality information, offers the possibility to envisage a pipelined execution managed by a dedicated skeleton construct. This represents a possible mean to exploit parallelism with existing efficient mechanisms. Such a mean is still in a study status. Solutions to recognize parallelism forms from an assembly and the possibility to map them on a skeleton constructs are required.

6. STKM EXPLOITED

In the Sections above, we have introduced STKM. In this Section we outline the key points and advantages of STKM by showing how two typical and significant use case applications can be implemented exploiting STKM methodology.

6.1 Fingerprint recognition in STKM

The first application we consider here is a refined version of a use case application considered in the framework of the GridCOMP EU STREP project [24]. In that context a fingerprint recognition application was considered that has to be able to match a fingerprint against a database possibly hosting a large number of fingerprints. The goal is to be able to get a real time answer telling whether or not the fingerprint is in the DB and, in positive case, the fingerprint owner identity [32]. In our extended version, we also consider the part of the application that collects fingerprints from real persons (e.g. at the airport arrival gates) and submits them to the fingerprint recognition software for processing.

Fingerprint matching against a DB can be nicely modeled using skeletons. This is a plain data parallel skeleton where parallel workers have been given a portion of the database and any single fingerprint is broadcast to all the workers. Referring to the *functional replication* behavioural skeleton as defined in [3], whose structure is drawn in Fig. 8, this corresponds to have identical worker components W specialized by submitting them different portions of the DB, a broadcast E port and a or-reduce C port (C gathers answers from all the workers and basically ORs the boolean values received).

Functional replication behavioural skeleton is one of the skeletons considered in STKM, and therefore this application can be easily expressed using STKM (Figure 9). Figure 10 illustrates the *spatial* aspects of the application. The left part handles gates, delivering requests to the $Check$ component. This component transforms requests issued on its provide port into items on the input stream for skeleton processing requests (the composite component in right part of the Figure) and conveniently returns the values received on its input stream port connected to the output of the recognition component as results of the provide port invocation. The upper part of

```
// port types are assumed to be defined
component FPApplication {

  component GateAdmin{
     uses CheckRequest uGA;
     ...Gate and MGR components...
  };

  funcRepl FPMatcher{
    inputSkel FPrint sInFPM;
    outputSkel string sOutFP;
    attribute  boolean batch;

    component Split {
       provides GetDB pDB;
       provides SetNbrW pW;
    };

    worker sequential cmpSkel {
       inputSkel FPrint sInCMP;
       outputSkel boolean sOutCMP;
       component cmp {
         provides SetDB pDB;
         streamIn FPrint sInCMP;
         streamOut boolean sOutCMP;
       };
    };

    instance Split sp;
    connect strmInCMPSkel to cmp.strmInCMP;
    connect cmp.strmOutCMP to strmOutCMPSkel;
    connect cmp.pDB to sp.pDB;
    emit-collec :: (broadcast, Or-reduce);
    sharedStateComp sp;
 };

  component Check {
     provides CheckRequest pC;
     streamOut FPrint sOutC;
     streamIn boolean sInC;
  }
  instance GateAdmin gateAd;
  intsance FPMatcher fpm;
  instance Check chk;

  connect chk.sOutC to fpm.sInFP;
  connect fpm.sOutFP to chk.sInC;
  connect gateAdmin.uGA to chk.pC;

  sequence ApplMain{
    exectask(fpm);
    exectask(chk);
    exectask(gateAd);
  };
}
```

Figure 9: Simplified STKM assembly for the Fingerprint recognition application example.

the Figure outlines the internal structure of the workers of the functional replication skeleton instance and of the Gate components. The former is a wrapping of the single fingerprint matcher (i.e. of the pre-existing component cmp that provides a port used to supply it the fingerprint DB, and two stream ports for accepting fingerprints to match and for delivering the corresponding answers) that eventually implements a provide port accepting "DB re-read" requests from the manager and a use port to access the DB portions in the $Split$ component. The latter is a standard loop initializing the gate, scanning a fingerprint, submitting it to the matching system and publishing the result of the match.

From the temporal viewpoint, the application components happen to be hosted in a sequence that first launches the FingerPrint matcher component, then the Check one and eventually the

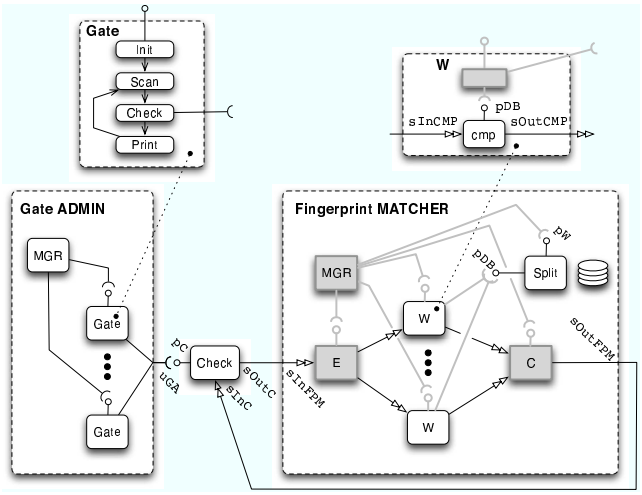


Figure 10: Spatial composition of the Fingerprint recognition application. The gray part is hidden to the designer.

GateAdmin manager. The STKM description of the sequence is shown in the last part of Figure 9. It is worth pointing out that exploiting skeletons, we can easily modify the FingerprintMatcher to process a huge amount of fingerprints in *batch mode*. In this case we can simply instantiate the functional replication skeleton in such a way the E port sends each input item to a different, "free" worker, C just gathers answers and delivers them to output and workers all receive (or access) a copy of the whole fingerprint database. Then, exploiting STCM derived workflow management, we can write an STKM program that depending on some input parameter from the system user activates either the "batch" or the "real time" matching composite component.

6.2 Climatology application in STKM

The second application we consider in this Section is a climatology application. It is basically a parameter sweeping application. For each parameter set, a number of iterations modeling climate evolution in the next 200 years is computed. Its structure is outlined in Figure 11 (a). The first component $S0$ is basically a component implementing a *forAll* construct. It iterates on the input parameter set sequence delivering a new parameter set to component $S1$. This, in turn, iterates computation performed by $S1$ to $S5$ for a number of times, in a sequential loop. Each iteration builds the approximate climate state at the next time quantum. Eventually, component $S5$ delivers the final result to component $S6$ for post-processing. Component $S4$ has a sensibly higher (10 times higher) execution time than the other components used in the application. This is a high level schema of a real application considered within the French ANR *LEGO* project [15].

Climatology experts having available all the components relative to the building blocks of the climatology application will probably come out with an application structure such as the one of Figure 11 (a). A component will provide the subsequent (in the temporal dimension) components with as much input items as the number of the parameter item in the input parameter set. By simply recognizing that the loop around components $S1$ to $S5$ is executed on a stream of input items, produced by component $S0$, and properly exploiting STKM, the application can be more or less "automatically" transformed into the one represented in Fig.11 (b). In

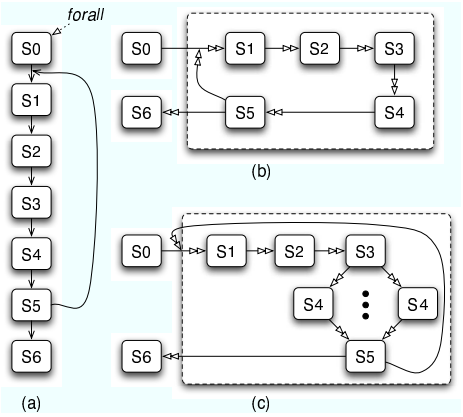


Figure 11: Example of a composition from which it is possible to recognize a pipeline.

this case, temporal composition of components $S1$ to $S5$ has been transformed into a spatial composition corresponding to a "loop of pipeline" skeleton composition, possibly exploiting wrappings such as those shown in Fig. 5. In turn, the new spatial composite deriving from the compilation of a loop of pipeline skeleton can be optimized much more than the original "temporal only" schema of Fig. 11. For instance, exploiting the estimated completion times of pipeline stages, stages $S1$ to $S3$ can be deployed within the same computational resource, preserving the service time of the loop of pipeline computation and, in the meanwhile, increasing the efficiency of the overall application. The net effect of using fewer resources can be estimated in passing from an efficiency around 20% to one above 80% (this looks huge, but actually, using one separate resource for each component in the application is quite an inefficient initial implementation). Alternatively, the application can be restructured as in Fig. 11 (c). In this case, the stage $S4$ has been parallelized by transforming the loop of pipeline in a loop of pipeline of farm, decreasing the service time of the overall pipeline and therefore increasing again the efficiency of the whole application. In this case efficiency can be obtained which is very close to 100%, due to the fact we can easily add 10 workers to the farm and therefore keep the service time of the "huge" $S4$ stage close to the service time of the other pipeline stages, and thus optimally balancing the whole pipeline (application).

It is worth pointing out two things here. First, the results above have been derived simply using well known performance models of pipeline and farm skeletons in conjunction with rough estimates of the time needed to compute component services and to communicate parameters among components. Previous experiences and experimental results achieved in the algorithmic skeleton frameworks completely validate this kind of reasoning. Second, none of the transformations/optimizations discussed above could have been implemented in the temporal only application specification (the one of Fig. 11 (a)).

6.3 STKM vs. standard approaches

We want to analyze in more detail the advantages of STKM against plain components, workflows and the original STCM; then, we qualitatively discussed the use case applications above. In particular, we consider the following properties of the programming model:

Expressiveness of an assembly the expressive power provided to the programmer to assembly applications out of their building blocks

Required designer expertise to implement efficient applications

Efficiency of the resulting assembly/application.

Tables 1 and 2 outline our judgment about the properties just stated in case of the fingerprint recognition applications (Table 1) and of the climatology application (Table 2). Just to understand how we compiled the Tables, let us detail how the "values" in column "designer expertise" of Table 1 has been determined. In case the fingerprint recognition application was to be implemented with a traditional component model, high programmer expertise is required if dynamic management of component composites are to be implemented such as those implemented by behavioural skeletons application managers. Even if workflows were used, programmer expertise required is high, as workflows do not support natively complex parallelism exploitation patterns such as the one present in the fingerprint application. Using STCM or skeleton systems, the programmer can use limited forms of parallelism (forAll in STCM, as an example) or limited (or null) temporal composition (workflow) support in skeletons, and therefore an average expertise is required to handle aspects not primitively supported by the environment (parallelism exploitation patterns in STCM and temporal composition in skeletons). STKM provides suitable mechanisms to handle all the modeling aspects of the fingerprint recognition application: temporal composition to handle skeleton and non skeleton component setup and skeletons to handle complex parallel pattern, possibly in autonomic way via the behavioural skeleton internal manager.

Both Tables evidence how STKM presents several advantages over the component, workflow and skeletons programming models.

7. CONCLUSIONS AND FUTURE WORKS

We outlined STKM, a programming model combining the advantages of components, workflows and algorithmic skeletons. Programmers can exploit workflow features of STKM to model applications in such a way the temporal relations between their different parts are precisely exposed, and they can also use skeletons to implement those parts of the applications that exploit parallelism according to well-known parallelism exploitation patterns. The environment exploits component technology, to allow programmers to implement applications by component assembly. In case of workflows, components are interconnected using new "temporal" ports, whereas skeletons are plain composite components whose inner components are interconnected by way of "stream" ports and their external interfaces also are based on stream ports.

We illustrated the feasibility of the STKM approach providing an extension of STCM (a model already supporting components and workflows) that includes common algorithmic skeleton. Using STKM, we modeled a couple of significant applications that happen to be use cases in distinct European projects. The STKM (abstract) version of the two applications allowed to outline the benefits of the approach as well as the added value with respect to STCM and the other component only, workflow only and skeleton only programming environments. In particular, we've shown how complex applications, can have parts that can be simply implemented exploiting skeletons (that is, instantiating one of the skeleton composite components provided by STKM) and inserted seamlessly in complex

	Expressiveness of an assembly	Level of designer expertise	Efficiency
Component models	average: synchronisation and dynamic management hidden in implementation	moderate (high for dynamic management)	high (static) expert level (dynamic)
Workflows	average: not captured construct	high	average: stateless (data transfer/reload)
STCM	average: enable to recognize some constructs	moderate (high for dynamic management)	proportional to expertize level
Skeletons	average: skeletons cooperation not natural	moderate (high when using non existing skeletons)	high
STKM	good	low	high

Table 1: Analysis of a the properties offered by different programming models to design the application represented in Figure 10.

	Expressiveness of an assembly	Level of designer expertise	Efficiency
Component models	hidden	high	proportional to expertize level
Workflows	average: adequate for temporal dependencies but often appears as a sequence	low	high: relies on global scheduler
STCM	average: adequate for temporal dependencies but appears as a sequence	low but designer has to use right ports	high
Skeletons	good	low	high: requires meta-data (execution durations)
STKM	good	low (for smart designer)	high: requires meta-data (execution durations)

Table 2: Designing a pipeline construct using different programming models. The analyzed example is shown in Figure 11 (part (a)).

workflows, and how, by exploiting skeletons in workflows, application implementation can be optimized.

While this paper focuses on theoretical background of STKM, future work considers its implementation and evaluation. To implement a component model such as STKM, several approaches can be followed. We are currently investigating an implementation on top of SCA (Service Component Architecture). The objective is to take benefit from the possibility to reuse an already existing model and, for the particular case of SCA, from the advantages of the underlying Service Oriented Architecture. The principle is to map an abstract representation of an application to an SCA architecture managed by an STKM engine. We also plan to have experiments validating the whole STKM approach even before the whole programming environment is implemented. In particular, we already implemented parts of the prototype applications considered in STCM and manually implementing skeleton composite components in such a way the combined usage of workflows and skeleton (in a component framework) can be evaluated and efficiency can be assessed as well. Preliminary experimental results achieved with these "hand programmed" experiments run on top of the SCA/Tuscany open source component/service programming environment demonstrate that the expected benefits related to the introduction of skeletons in STCM are actually there [1].

Acknowledgements

This work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265), within the framework of its Researcher Exchange Program no. 25. The work is also partially supported by the FP6 GridCOMP project funded by the European Commission (Contract FP6-034442) and the French National Agency for Research project LEGO (ANR-05-CIGC-11).

8. REFERENCES

[1] M. Aldinucci, H. Bouziane, M. Danelutto, and C. Perez. Towards a Spatio-Temporal sKeleton Model implementation on top of SCA. Technical Report TR-171, CoreGRID, 2008. available at `www.coregrid.net`.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance grid programming in grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 19–38, Saint-Malo, France, Jan. 2005. Springer.

[3] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In D. E. Baz, J. Bourgeois, and F. Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, Feb. 2008. IEEE.

[4] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006.

[5] M. Aldinucci and M. Danelutto. Securing skeletal systems with limited performance penalty: the Muskel experience. *Journal of Systems Architecture*, 54(9):868–876, Sept. 2008.

[6] 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 Intl. Workshop on Scientific Applications of Grid Computing (SAG'04))*, pages 120–133, Berlin/Heidelberg, 2005. Springer.

[7] 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.

[8] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

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

[10] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of the 11th Intl. Euro-Par Conference*, volume 3648 of *LNCS*, pages 761–770, Lisboa, Portugal, Aug. 2005. Springer.

[11] 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.

[12] C. Bertolli, M. Coppola, and C. Zoccolo. The co-replication methodology and its application to structure d parallel programs. In *CompFrame '07: Proc. of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 39–48, New York, NY, USA, Oct. 2007. ACM.

[13] H. Bouziane, C. Pérez, N. Currle-Linde, and M. Resch. A software component-based description of the segl runtime architecture. In *CoreGrid integration workshop 2006*, pages 69–80, Krakow, Poland, 19-20 October 2006.

[14] H. Bouziane, C. Pérez, and T. Priol. A software component model with spatial and temporal compositions for grid infrastructures. In *Proc. of the 14th Intl. Euro-Par Conference*, volume 5168, pages 698–708, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[15] CERFACS - European Centre for Research and Advanced Training in Scientific Computation. `http://www.cerfacs.fr/`.

[16] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[17] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[18] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, Feb. 2007. `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[19] M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[20] M. Danelutto and G. Zoppi. Behavioural skeletons meeting services. In *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, volume 5101 of *LNCS*, pages 146–153, Krakow, Poland, June 2008. Springer.

[21] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *PARLE'93 – Parallel Architectures and Languages Europe*, volume 694 of *LNCS*, pages 146–160. Springer, 1993.

[22] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proc. of the 5th IEEE International Symposium on Cluster Computing and Grid 2005 (CCGrid 2005)*, volume 2, pages 676–685, Cardiff, UK, May 2005.

[23] S. Gorlatch and J. Duennweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer, 2005. selected works from Dagstuhl 2005 FGG workshop.

[24] GridCOMP Project. Grid Programming with Components, An Advanced Component Platform for an Effective Invisible Grid, 2008. `http://gridcomp.ercim.org`.

[25] H. Kuchen. A skeleton library. In *Proc. of 8th Intl. Euro-Par Conference*, volume 2400 of *LNCS*, pages 620–629. Springer, August 2002.

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

[27] OMG. Unified modeling language. Document formal/2007-02-05, Feb. 2007.

[28] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[29] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, second edition, 2002.

[30] 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.

[31] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.

[32] T. Weigold, P. Buhler, J. Thiyagalingam, A. Basukoski, and V. Getov. Advanced grid programming with components: A biometric identification case study. In *Proc. of the 32nd Intl. Computer Software and Applications Conference (COMPSAC)*, Turku, Finland, August 2008. IEEE.

[33] 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.