European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

Towards a Spatio-Temporal sKeleton Model Implementation on top of SCA

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

Hinde Lilia Bouziane and Christian Pérez
INRIA/IRISA, Campus de Beaulieu
35042 Rennes cedex, France
{Hinde.Bouziane,Christian.Perez}@inria.fr

CoreGRID Technical Report
Number TR-0171
August 31st, 2008

Institute on Programming Model

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Towards a Spatio-Temporal sKeleton Model Implementation on top of SCA

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

Hinde Lilia Bouziane and Christian Pérez
INRIA/IRISA, Campus de Beaulieu
35042 Rennes cedex, France
{Hinde.Bouziane,Christian.Perez}@inria.fr

CoreGRID TR-0171

August 31st, 2008

Abstract

This report investigates an implementation of STKM, a Spatio-Temporal sKeleton Model. STKM expands the Grid Component Model (GCM) with an innovative programmable approach to compose an application by combining component, workflow and skeleton concepts. We explore a projection of the model on top of SCA and its implementation using Tuscany Java SCA. Experimental results show the need and benefits of the high level of abstraction offered by STKM.

1 Introduction

Many programming models are proposed to develop large-scale distributed scientific applications. They attempt to offer means to deal with the increasing complexity of such applications as well as the complexity of execution resources, like Grids. They also attempt to ensure efficient execution and resource usage. However, existing models often target different properties and/or specific kind of applications, usually determined by the usage of a given programming paradigm. A current challenge is still to offer a suitable programming model to easily and efficiently support multi-paradigm applications.

In this report, we focus on three well-known families of programming models: component, workflow and skeleton based models. They all follow an assembly/composition programming principle, which is becoming a widely accepted methodology to cope with the complexity of the design of parallel and distributed scientific applications. Component models mainly deal with code reuse problem; they are quite appropriate for strongly coupled compositions. Workflow models make it possible to establish temporal dependencies among components, thus to enable the efficient scheduling of components onto resources (e.g. sites, processors, memories). Components arranged in a workflow are typically loosely coupled. Eventually, algorithmic skeletons are suited to describe component assemblies in a fairly abstract way (e.g. high-order and parametric component assemblies). This enable designer to leverage on automatic optimizations for efficient execution on targeted resources [10].

In summary, each family of models has been considered suitable to deal with a class of problems affecting the programming of complex applications; these classes have been independently studied. Nevertheless, all these properties

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

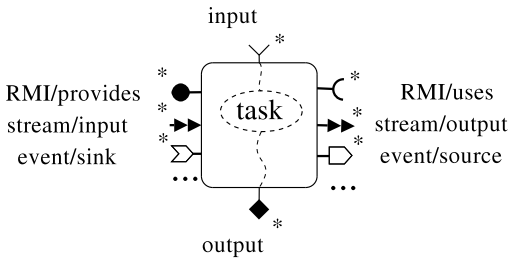

Figure 1: An STKM component.

seem to be relevant to be considered in a single programming model. In this work we aim to combine these families of models.

There are previous works that aim to combine these families. In particular, STCM (Spatio-Temporal Component Model) [9] is a model combining component models and workflows. Efforts have been also done to combine skeletons and component models [3, 11]. In a similar direction, STKM [1, 2] is an attempt to combine STCM with skeleton models. While previous work related to STKM explored theoretical background, this report investigates its implementation. In particular, we explore the possibility to allow users to design an application as the composition of extended GCM components (primitives and composites) and to use SOA/WS based framework to build an implementation. The extension of a GCM component considers the integration of workflow concepts (tasks and input/output ports) and skeleton constructs, such as pipeline and farm. All these concepts are built on top of SCA (Service Component Architecture). This report describes this perspective and discusses experimental results.

The rest of this report is organized as follows. Section 2 recalls the theoretical background of STKM. Section 3 introduces the followed approach to implement STKM on top of SCA. In particular, it describes how components "à la GCM" are projected to SCA ones and how an assembly is handled. Section 4 discusses the usage of Tuscany Java SCA to realize this implementation. The accent is put on the limitations of the used Tuscany framework that is still in development. Thus, preliminary solutions are proposed to respond to some STKM requirements. Section 5 discusses experimental results. It illustrates the feasibility of STKM concepts and the benefits of this model regarding the ability of the proposal to simplify programming and to automatically adapt an application to its execution context. Section 6 concludes the paper and presents future works.

2 Overview of STKM

In [1], we proposed to extend STCM [9] (Spatio-Temporal Component Model) with (behavioral) skeleton support. The advocated idea considers two issues. Firstly, a programming model should offer means to explicitly express the functional behavior of an application through its assembly. That is to promote simplicity of design and separation of functional concerns from non-functional ones (example: component life cycle management, processes management for parallel codes). Secondly, such non-functional concerns are expected to be transparently managed by the component framework. The level of expressiveness of an assembly is relevant to enable a framework to adapt an application to an execution context (execution resources) and ensure its portability to different contexts. Thus, STCM offers a level of abstraction allowing a designer to express both temporal logic of an application execution (inherited from workflow models) and spatial dependencies between components (inherited from component assembly models). STKM adds the possibility to use predefined skeleton forms to build parallel composition of codes. This section recalls the principle of STKM putting the accent on skeletons support. In particular, it describes the unit of composition of an STKM application, its assembly model and the suitable approach to manage the assembly by the framework.

Component. STKM reuses STCM components. Thus, a component, originally named *component-task* in STCM, is a combination of component and task concepts. As shown in Figure 1, a component can expose *spatial* and/or *temporal* ports. Spatial ports are classical component ports. Temporal ports (input/output) and tasks behave like in a workflow. The difference is that the life-cycle of an STKM component may be longer than the one of a task in a classical

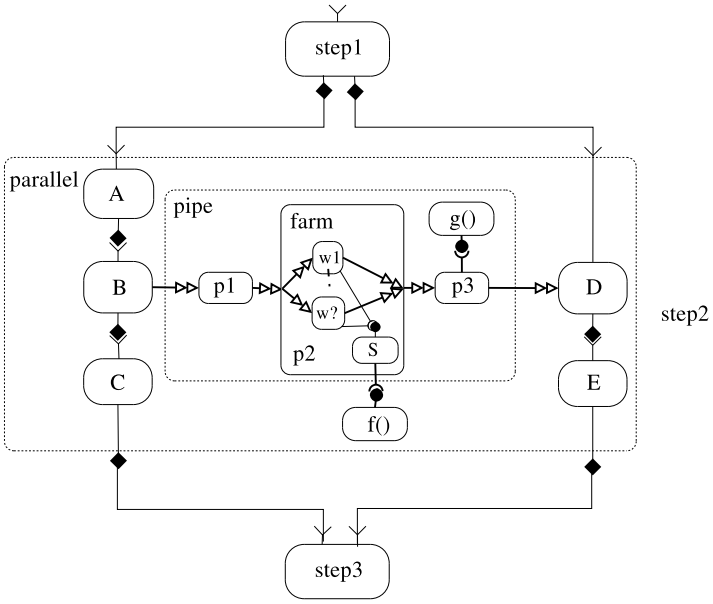

Figure 2: Example of an STKM assembly.

workflow, which usually corresponds to its execution. More details about the specification of such components can be found in [9]. This specification is presented through an extension of a GCM (Grid Component Model) component.

Assembly model. An STKM assembly is first of all a combination of temporal and spacial dependences between components (primitive or composite). Spatial dependencies are represented by the connections of spatial ports, as in classical component models (such as GCM). Temporal dependencies are represented by both a data flow and a control flow as done in a workflow. A data flow is built by connecting input and output ports of dependent tasks. A control flow is built by using control constructs like sequences, branches (*if* and *switch*), loops (*for* and *while*) and parallel constructs (*parallelFor* and *parallelForEach*), etc. STKM adds constructs dedicated to skeleton-based parallel paradigms. In more details, these constructs are particular composite components (templates) representing skeleton schemes (pipe, farm, functional replication, etc.). The internal structure of such a component is well defined according to a given parametric connection schema. A skeleton can be composed with components and/or other skeletons. That can be done at different levels of an assembly. Also, skeletons can be nested. These possibilities improve composability and code reuse, while preserving the pragmatic of skeletons.

Figure 2 illustrates an example of assembly supported in STKM. It describes a sequence of three tasks wrapped by components `step1, step2` and `step3`. The component `step2` is a composition of two parallel sequences `(A;B;C)` and `(D;E)`. The second sequence depends on the result of pipelined stages (`p1, p2, p3`) for which inputs are produced by component `B` of the first sequence. The composition also illustrates nested skeletons (`farm` as stage in the `pipe`). Moreover, the figure shows that skeleton elements (stages of a pipe or workers in a farm) have the possibility to express dependencies with other components. This may be exploited, for instance, to express a shared state. Details about the STKM assembly language, the semantic of an assembly as well as an example of its usage for a real world application can be found in [1].

Assembly management. As addressed at the beginning of this section, the objective of STKM is not limited to simplify the design of applications. It aims also to take benefits from the expressiveness power of an assembly to efficiently execute an application on given resources. The efficiency essentially depends on the ability of a framework to exploit the maximum parallelism from a part or the whole assembly, and to adopt an adequate scheduling policy depending on actually available execution resources. In this direction, STKM proposes to consider parallelism forms that are explicitly expressed by using skeleton constructs as well as those built without using skeletons but which can be mapped to a skeleton composition. An example for the last situation is to map the independent `forAll` control

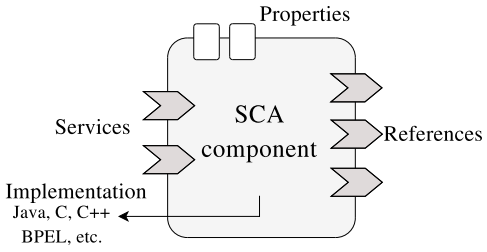

Figure 3: An SCA component.

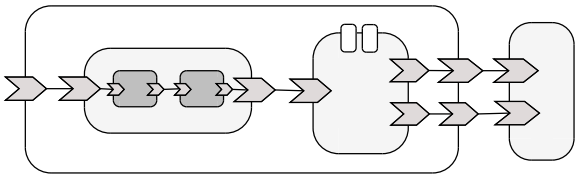

Figure 4: Example of SCA assembly.

construct to a functional replication skeleton in which the workers are the body of the loop [1]. When the mapping is done, skeletons constructs have to be projected to concrete implementations. In STKM, it is at the responsibility of the framework to decide an adequate implementation depending on the execution context. The approach is to reuse already proposed component based implementations, like those based on GCM components [3, 13]. Such implementations offers self adaptive management support for computational elements and are able to deal with optimization issues for instance collapsing stages of pipes or introducing farms for efficiency. Therefore, STKM should take benefits from already existing skeleton management mechanisms able to deal with performance [4] concern as well as other like security [5] or fault tolerance [8].

3 An approach to implement STKM concepts on top of SCA

In this section, we discuss an approach aimed at implementing STKM using SCA. Our main purpose was to verify the feasibility of STKM concepts. While STCM was originally thought as an extension of GCM, we envisioned it was worth to explore the feasibility of STKM in a *Service Oriented Architecture* (SOA). On the one hand, this will eventually allow to compare the STCM "temporal" part (i.e. the workflow one) with plain services as used in the implementation of existing service workflow frameworks and environments. On the other hand, by using SCA we wanted to investigate whether porting STCM/STKM concepts to the service world is as effective as porting other GCM concepts, as already demonstrated in [11].

In order to implement STKM on SCA, two distinct issues must be taken into account: a first issue is related to the projection of the user view of an STKM assembly to an SCA assembly, while the second one deals with the management of an application assembly during an application execution. Before discussing how we can deal with these two issues, let us give an overview of SCA.

3.1 Overview of SCA

SCA [7] is developed since 2005 by the *OSOA* (*Open Service Oriented Architecture*) group. It defines a specification for programing applications according to a *Service Oriented Architecture* (*SOA*). The objective is to enable composition of services independently from the technologies used to implement these services and from any SCA compliant platform.

The SCA specification deals with several aspects: assembly, client and component implementation, packaging and deployment:

Assembly model. In this model, a component is defined as a set of ports named *services* and *references* (Figure 3). Ports are of several kinds depending on the technology used to implement a component. They may be interfaces of type Java, IDL CORBA, WSDL, etc. They allow interactions between two connected components based on message passing, Web Services or RPC/RMI communications. The interoperability between components implemented using different technologies is ensured through the specification of dedicated binding mechanisms. A component can also define *properties* to specify configurable attributes. An SCA assembly (Figure 4) can be hierarchic. Compared with GCM, the hierarchy is abstract. It is used to determine the visibility frontier of sub-components of a composite component. Then, ports of a composite are representation of sub-component ports to be exposed outside. The membrane concept, as existing in GCM, is not existing in SCA. Therefore, the objective in SCA is limited to preserving encapsulation and simplifying assembly process.

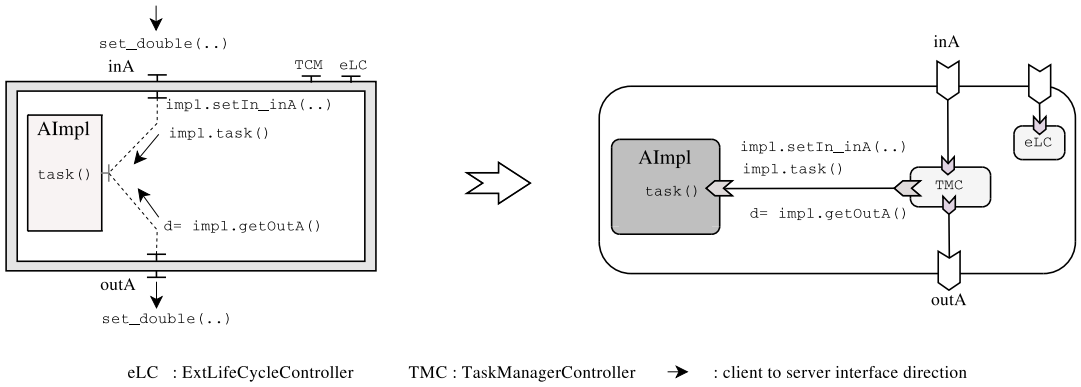

Figure 5: Example of mapping an STKM component to an SCA one.

Client and implementation model. SCA defines for each implementation language a specification for service implementations. This specification describes the SCA model for a given language independently from any SCA platform. For that several means are used like annotations for Java/C++ or XML extensions for BPEL (Business Process Execution Language). These means permit for instance the definition of services, properties, and meta-information like local or remote access constraints associated to a service.

Packaging and deployment model. This part focuses on the specification of component packaging. It describes the unit of deployment associated to a component. For deployment concerns, an SCA platform is free to define its proper model. Section 4 gives an overview of the model realized by Tuscany Java and used to perform presented experiments.

3.2 A projection of STKM concepts on top of SCA

To implement STKM on top of an SCA framework a projection of STKM concepts to SCA ones is required. These concepts are components and assembly. This section describes our projection approach for these two concepts.

3.2.1 From an STKM component to an SCA component

As mentioned in Section 2, the user view of STKM considers components as an extension of GCM components. Figure 5 gives an overview of the extension approach we adopted. Only temporal ports and tasks concepts are concerned. The left part describes the projection of task and temporal ports to classical GCM concepts. A task is just an implementation of a server interface. Temporal ports are mapped to classical GCM client/server ones. The implementation of these ports is realized by a dedicated controller, which is responsible to manage input and output data availability and execution of tasks. Hence, this management is transparent for the user. The right part of the figure maps a GCM component to an SCA one. A GCM component is mapped to an SCA composite component, GCM ports to RPC/RMI services/references, controllers to SCA components and sub-components to SCA component implementation. Note that the representation of controllers by components is not a new idea. This principle promotes composability and code reuse and it is well accepted in the GCM specification. Also, there exist works attempting to build a GCM implementation using SCA [11]. The principle is similar to the one presented here.

The projection of STKM components also considers skeleton constructs. As these constructs are components, a similar approach is followed. The difference is that a concrete representation of a skeleton may consider additional non-functional elements like managers for behavioral skeletons. Figure 6 shows a representation example for a functional replication skeleton. Components `MGR` (manager), `E` (emitter) and `C` (collectors) are the non-functional elements of the skeleton. They are expected to be transparently introduced in the skeleton construct by an STKM framework. Note that the whole structure can be directly realized by SCA components or can be first mapped to an STKM assembly and then projected to an SCA one. This possibility allows reusing existing implementations of skeleton constructs realized for instance with GCM [6].

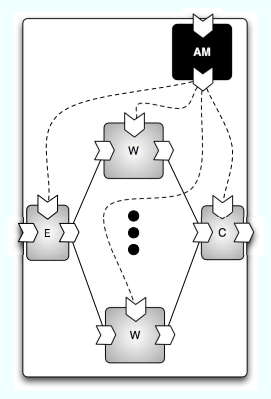

Figure 6: A concrete representation of a functional replication behavioral skeleton component.

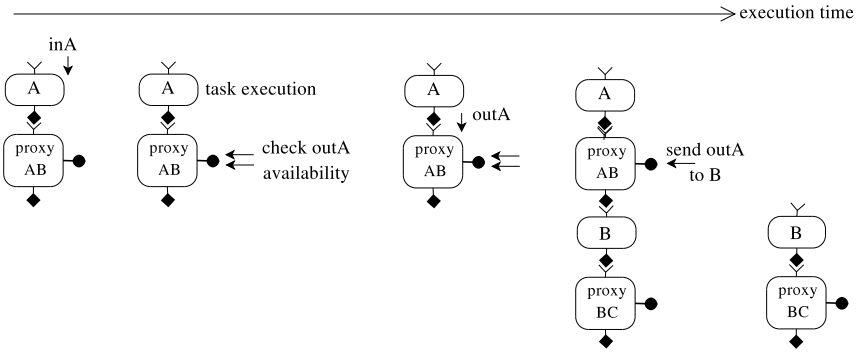

Figure 7: Data transfer management through a proxy component. The STKM notation for ports is still used here for simplicity. Concretely, components are SCA ones according to the direct mapping shown in Figure 5.

To summarize, a simple projection of STKM components to SCA components was possible. That is thanks to the support of RPC/RMI ports by the two models, their hierarchical property and the possibility to realize controllers by components.

3.2.2 From STKM to SCA assembly and its management

In STKM, an assembly is not limited to component instances and spatial connections. Temporal dependences may be also specified by data and control flow constructs offered by STCM. Thus, it is not sufficient to project temporal ports to SCA ports, as described in Section 3.2.1, to be able to perform a direct projection of an STKM assembly to an SCA one. In fact, one of the objectives of STKM/STCM aims to dynamically modify the structure of an application according to both specified temporal and spatial dependencies. For that, we propose a projection to an SCA assembly for which the structure is dynamically and automatically modified during execution.

In this context, the issue is to introduce mechanisms to manage data transfer between dependent components, the sequence of tasks executions according to both designed data flow and control flow and the life cycle of components. Several solutions can be proposed. In this report, we present a preliminary adopted solution, based on a distributed data transfer approach and a centralized engine “à la workflow”. The remainder of this section gives an overview of our proposal.

Data management There are two alternatives to transfer data between two components connected within a data flow. These alternatives depend on the co-existence of these components. In fact, if the components co-exist, it is sufficient to connect the output port of the first component to the input port of the second one. A typical situation is when the tasks of the components are successive pipeline stages. However, if the components do not co-exist, a mechanism is needed to retrieve a produced data and subsequently send it to a next component. For that, we propose to introduce

```
create component  A;
create component  proxyAB;
connect A to proxyAB;
order send input on inA;

check data availability on proxyAB;
while not available
  check data availability on proxyAB;

create component B;
create component  proxyB..;
connect B to proxyB..;
order proxyAB to send data to B;
remove A;
remove proxyAB;
check data availability on proxyB..;
...
```

Figure 8: Simplified STKM engine managing the sequence shown in Figure 7.

dedicated proxy components in the assembly at execution. Figure 7 illustrates this proposal for a typical situation of a sequence. When the task of component A produces a data, this latter is send to the proxy component. When component B is created, it is connected to the proxy. The data is sent to B when the proxy receives a corresponding request (cf. STKM engine paragraph). The figure also shows the evolution of the assembly according to the evolution of tasks executions. The specification and the implementation of a proxy are assumed to be done automatically. A proxy is viewed as a template component for data transfer. The parameters of such a template are input and output port types for a specific usage context. These types are the ones defined by user level components. The introduction of a proxy in the assembly is assumed to be done by an STKM assembly interpreter. This interpreter is responsible to perform the STKM assembly projection to its concrete SCA based representation.

STKM engine The STKM engine appears as an SCA client program. This program is the result of the STKM assembly interpreter. It contains the sequence of actions to create/destroy components, connect/disconnect component ports, manage data availability/transfer and cover the control flow described in STKM assembly for ordering tasks execution. These actions are deduced from the behavior expressed by the STKM assembly. Figure 8 shows a simple engine example managing the sequence shown in Figure 7. In this example, it is assumed that a component is created when the control flow reaches it. After an input data is sent to component A, the engine waits the end of its task execution. For that, it checks the output data availability of A on `proxyAB`. Once the data is available, the engine orders the creation of components B and `proxyBC`. Then, the engine orders `proxyAB` to send the saved data to B. We recall that it is not at the responsibility of the engine to directly manage a task execution. This is delegated to the component controller. In general, this approach promotes a distributed management and should simplify the engine role. However, the illustrated example adopts a scheduling approach that may lead to scheduling overheads. In fact, the creation and configuration of B and `proxyBC` is done only once the task being executed on A is finished. Other solutions may be adopted to overlap the creation/configuration of components with computation. Such solutions may be based for instance on prediction mechanisms. Without ignoring the relevance of using an efficient scheduling approach, this report does not study such solutions.

The presented approach deals with an STKM assembly at execution, including projection on top of SCA and the STKM engine, is not conflicting with the objectives of STKM. All concerns encountered in this section are non-functional and are hidden to the user. This later may then keep the simple view of an application. The remainder of this report treats performance obtained using an SCA framework.

4 Usage of Tuscany Java SCA

To evaluate the work presented in this report, we realized an implementation for STKM use cases. For that, we used the Tuscany Java SCA framework Version 1.2.1 [14]. This framework is under development. It realizes part of the SCA Java component implementation specification (Version 1.0) and provides a preliminary support for distributed execution of applications. This framework provides no deployment tools. This section gives an overview of the

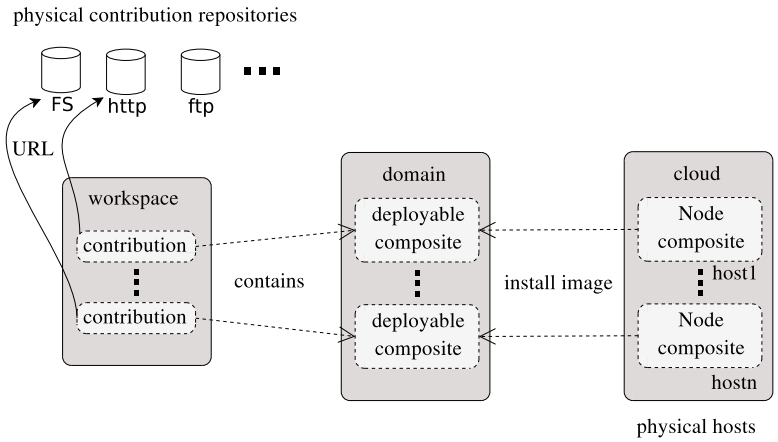

Figure 9: Distributed execution environment of an SCA application using Tuscany Java framework.

distribution principle of the framework, the limitations evidenced when using SCA for implementing the STKM support as well as the solutions adopted to overcome such limitations.

4.1 Distribution principle

Figure 9 describes the main concepts of a SCA distributed execution environment. In this environment, the domain is an administrative concept. It is responsible to build and resolve an image reflecting installed contributions. A contribution is an archive containing artifacts required to deploy a component (JAR or ZIP files) [7]. The cloud composite provides a global image of all deployed nodes. A node is a particular top-level composite component which hosts specified components in a same process. The deployment of components is manually done by launching the hosting node on a given machine. While the effective instantiation of a component within a node, it is done in a lazy way (at the first service invocation on the component). Finally, the entry point to execute an application is a client program. An API is offered to such a program to access services provided by components.

4.2 Limitations and adopted solutions

As we cited at the beginning of this section, the support of the SCA specification in Tuscany is still in development. Some features are not yet supported or ported from previous Tuscany distributions. In particular, the distribution presents lacks regarding the support of components/services lookup, dynamicity, and some binding protocols in a distributed context. Also, some problems were encountered when executing nodes in a cluster. Let us briefly detail each limitation and the adopted solutions with respect to STKM requirements:

Component and service lookup SCA describes an API allowing SCA services to be programmatically accessed by client programs which are not running as SCA components. However, this API is not implemented in the used framework. Actually, the access to a component service by a client is done through a node executed in the same space as the client. In the context of STKM, we followed this same principle for an STKM engine which, as seen in Section 3.2.2, is a client program. Without loss of generality, Figure 10 illustrates this principle for accessing a proxy port. As it can be noted, an indirection is generated. Its cost is equivalent to an intra-process method call which does not affect discussed experimental results presented in this report.

Dynamic addition/removal of components Dynamic assembly modification capability is not completely and/or efficiently treated in the used framework. In fact, to dynamically add/remove components, there are mainly two approaches. The first approach is based on dynamic addition/removal of contributions and nodes. Its principle allows dynamic specification of new components as well as nodes with different configurations. However, this approach is

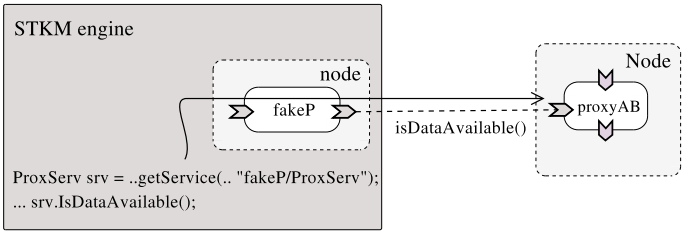

Figure 10: Access to a component service from an STKM engine.

```
// This interface is implemented by
// the controller TMC (Figure~5)
public interface A_Ctrl_Ports {

  //input side (inA)
  public String provides_inA();

  //output side (outA)
  boolean connect_outA(String srvRef);
  public void disconnect_outA();
}
```

Figure 11: Usage of service reference passing in SCA to specify ports connection/disconnection operations. The

not well supported in the used Tuscany framework. It also requires domain reconfiguration that currently needs stopping an application at each reconfiguration[1]. The second approach is based on the addition/removal of nodes only. Alternatively with respect to this former approach, this second approach does not require to suspend an application execution when reconfiguring it. Its limitation however is that all nodes are statically defined and configured. Components hosted by nodes as well as hosting machines are known in advance. Even if at the end our objective is to support dynamic decisions, this limitation does not affect the objectives of performed experiments. We then decided to make static decisions and follow the second approach for the present work.

Dynamic connection/disconnection of ports SCA does not explicitly provide user API to programmatically connect/disconnect references to services. However, it provides an API to allow passing service references. Without lost of generality, we used such an API to specify connection/disconnection operations for temporal ports. Figure 11 presents defined operations for input and output sides. All operations are exposed by corresponding services. The specification and implementation of these services affects non-functional part of components, expected to be transparently generated. In this context, we faced an additional limitation of Tuscany. This limitation is related to the support of different kind of binding protocols associated to services passed by references (`srvRef` in the example). In other words, it should be sufficient to configure such services with a default SCA protocol. However, that is currently missing. Hence, we replaced this protocol with a Web service one. As will be presented in Section 5, this protocol affects the communication time between two components. It should be also taken into account that a serialization a service reference was needed for its passing (`String` type in the Figure).

Node launching on clusters Tuscany distribution provides node launcher programs to execute node process as detached daemon on Linux (the operating system used for our experiments). However, such an execution on the used cluster causes Input/Output failures. Hence, we developed our own node launchers. For that, we used the common-daemon library [12] which provides a support to make the interface between the daemon to the operating system.

To sum up, we resolved current limitations of used Tuscany framework with respect to some STKM requirements and without modifying it. In next Section, some presented results are affected by these limitations as well as proposed solutions. However, it is to note that this is sufficient to illustrate the benefits of STKM and its feasibility on a Web Service based environment.

5 Evaluation

STKM aims at increasing the level of abstraction with respect to parallel programming and offering a powerful management of an application execution. This section evaluates the performance of the model that may be obtained by using a Web Service environment for various and simple situations. More precisely, we developed an application

[1] It is relevant to note that efforts are done to overcome this limitation. A solution is proposed in [11, 6] and its feasibility was proved using an old Tuscany version.

	Remote note Launching time	Programmatically connecting two components
Time in s	45.555	3.204

Figure 12: Average of times to deploy and connect components.

RTT in ms	Intra-node Inter-component	Inter-Node Intra-host	Inter-Node Inter-host
Default protocol	0.076	20.348	20.167
WS protocol	22.664	24.225	24.106

Figure 13: RTT on a local Ethernet network for different situations.

according to different compositions: sequence, pipeline and nested composition of pipeline and farm/functional replication skeleton constructs. As discussed in this section, the deployment and execution of this application in different execution contexts will clearly outline the benefits of STKM.

All experiments have been deployed and executed on a cluster made of 24 Intel Pentium 3 at 800 MHz PEs, 1 GB RAM, 4200 rpm disk, connected through a 100 MBit/s switched Ethernet and running on Linux (2.4.18 kernel).

The conversion from an abstract STKM assembly to SCA components and STKM engine was done manually. Components are implemented using Tuscany Java SCA version 1.2.2. All code is is written in Java 1.5. Then, the deployment of components was done at the initiative of the engine by using `ssh`.

5.1 Metrics

The first question is to evaluate basic overheads which an application execution may have according to dynamic life cycle management of components, communication overheads between components within the Tuscany environment as well as the effect of communication protocols on communication times. The measure of such overheads helps understanding next discussed performance results.

Figure 12 illustrates first metrics. The first column shows obtained results for deploying and launching a node. As it can be noted, starting a node is costly. The obtained time covers Tuscany proper creation of nodes policy (based on the usage of class loaders) and the usage of common-daemon library (Section 4.2) to launch a node process. The second column shows obtained results for connecting ports through the reference passing mechanism explained in Section 4.2. The result is mainly a consequence of serialization/deserialization needed by this operation. It is relevant to recall that part of the overheads are due to limitations of used framework and added overheads of preliminary adopted solutions. Improving Tuscany as well as resolving its limitations should improve these metrics.

Figures 13 reports the round trip time for used services/references configurations and for different placement of components. The RTT time corresponds to an empty service invocation. The configuration of a service specifies the possibility to remotely access a service and the binding protocol used to invoke it. We used default SCA protocol and Web Service protocol. This latter is used for services passed by reference (Section 4.2). In the context of the present work, it concerns services associated to temporal ports. In the figure, all services are configured to be remotable except for the first column. Several conclusions can be drawn. First, modifying the binding protocol affects communication times even between components within a same process (First column). That may be explained by a lack of communication optimizations in the current Tuscany environment. Second, the impact of the network is negligible, even hidden, with respect to the effect of communication protocols. For a remotable service (configuration used for inter-node and inter-host placement), the effect of changing a protocol is not relevant. The RTT, around $20 - 24$ ms, in all cases lead to costly communications. That is not a surprise as SCA specification addresses this issue and claims that SCA is more adequate for coarse grain codes.

5.2 A sequence use case

The second experiment illustrates the consequence of mapping an STKM assembly to different concrete assemblies as well as of components placement on the overall execution performance that may be obtained for a simple use case.

Figure 15 reports the execution time of a sequence of 4 tasks according to several configurations. These configurations are introduced in Figure 14. They represent two concrete assemblies for a same experimented application. For the

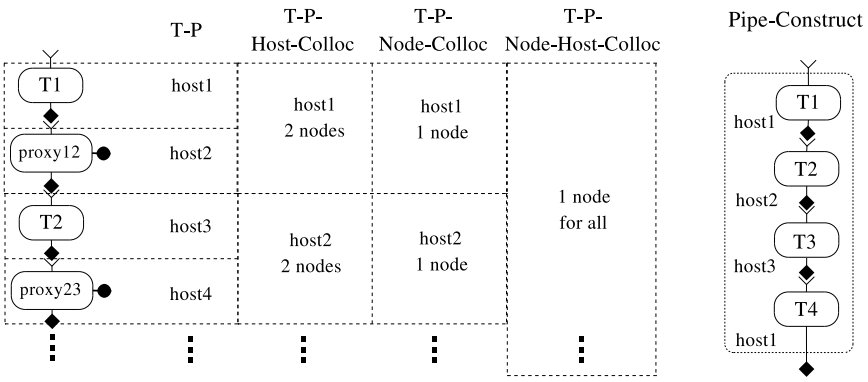

Figure 14: Configurations for the results shown in Figure 15. T: Task in component, P: Proxy, Colloc: Collocation.

	Global time (s)	Computation (%)
T-P	295.966	27.030
T-P-Host-Colloc	397.347	20.134
T-P-Node-Colloc	281.878	28.381
T-P-Node-Host-Colloc	280.707	28.499
Pipe-Construct	156.421	51.144

Figure 15: Time to execute a sequence of 4 tasks. The execution time of each task is $20s$. The life cycle of a component-task is delimited by input/output data availability.

assembly on the left part, the execution and life cycle management of components follows the principle presented in Section 3.2.2. In this case, several components placements are experimented. On the right part, the original sequence (user level) is mapped to a pipeline composition. In this case, all implicated components are deployed and connected by the engine before starting the first task execution. Components are deployed on different machines, as usually done for pipeline constructs in general. Without surprise, the `Pipe-Construct` configuration lead to more efficient execution. Note that the reported result includes remote nodes creation ($\simeq 6s$ for executing remote commands and $45s$ for waiting starting all nodes), components instantiation and port connections ($\simeq 17s$). Even if measured metrics may be improved, they are expected to have similar impact with less cost on the wall application execution time. Such an impact should become negligible for course grained codes.

However, even if `Pipe-Construct` provides better performance, it is relevant to take into account other criteria in the choice of a concrete assembly, in particular, resources usage. In fact, `Pipe-Construct` may causes an overconsumption of resources. That may be problematic when using infrastructures like Grids because of shared resources. While other configurations should offer the ability to optimize resources usage with efficient scheduling policies. For the present work, the objective is not to study such policies. This is why we compared only basic configurations testing the behavior when modifying the deployment of components (Figure 15). Results show that the `T-P-Node-Host-Colloc` configuration provides better results. That may be explained by the fact that all components are executed in the same process. Also, the lazy instantiation of components in Tuscany framework reduces the number of threads executed simultaneously. However, components in a sequence may requires different processor with respect to eventual execution constraints/contracts like a particular operating system, a minimum memory space or processor speed, etc. Moreover, the availability of used resources during the execution time should be considered. Therefore, a configuration that simply allow a placement on different resources is suitable. For that, we selected the configuration `T-P-Node-Colloc`. This is why in the remainder of this report the other configurations, `T-P`, `T-P-Host-Colloc` and `T-P-Node-Host-Colloc`, are not reused.

Finally, the advantage of STKM is its ability to choose a concrete assembly according to a given execution context. That is without changing the high level assembly of the application.

	Global time (s)	Computation (%)
nbr data 10		
Loop	2817.227	28.397
Loop-Opt	462.949	56.162
Pipe	337.860	76.955
nbr data 100		
Loop	28167.841	28.401
Loop-Opt	2271.786	90.677
Pipe	2147.984	95.904

Figure 16: Effect of using different constructs/life cycle management on the execution time of a sequence of 4 tasks on multiple input data. Execution time of a each task is $20s$.

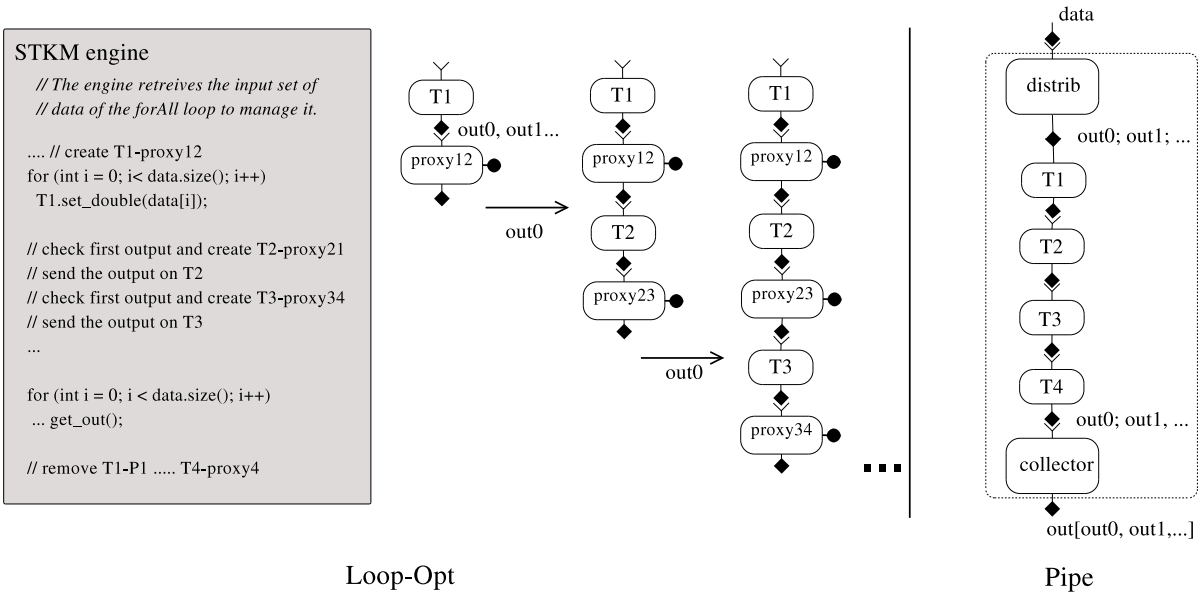

Figure 17: Overview of the two used configurations for executing an independent *ForAll* loop

5.3 Need of forms recognition

STKM promotes the ability of an STKM engine to recognize parallelism forms from an assembly and to exploit them in the generation of a concrete assembly for efficient execution. STKM encourages also a mapping of parallelism forms to a composition of skeleton constructs. This section illustrates the interest of the underlying idea through a use case of an independent *forAll* loop.

Experiments are done for executing a *forAll* loop using three different concrete assemblies. The body of the loop is the sequence composition experimented in Section 5.2. The concrete assemblies are represented by the three configurations listed in the first column of Figure 16. `Loop` configuration means that no parallelism form is recognized. Each iteration considers one data and all iterations are sequentially executed. The life cycle of component instances is managed as done for a sequence (Figure 7) for each iteration. For both `Loop-Opt` and `Pipe` configurations we exploit the parallelism form of the *ForAll* loop to support parallel execution. An overview of the followed approach in the two configurations is shown in Figure 17. In both configurations, the execution is based on pipelined computations of the `ForAll` input data. However, they differ on the way the concrete assembly is managed:

- `Loop-Opt`: In this configuration, the *ForAll* control structure is implemented by the STKM engine. It is at its responsibility to split and collect the input data of the loop. For life cycle, a component `Ti` is created once the first output of `Ti-1` is available. Created instances are removed only after the loop execution. Even if a pipeline execution is built, proxy components remain present as they are part of the sequence mapping. Note

	Global time (s)	Computation (%)
Without load balancing	2116.355	95.920
Node-Colloc for step 2 and 3	2113.625	96.043

Figure 18: Load balancing for executing pipelined 4 tasks. The execution time of each task is (in order): $10s$, $15s$, $5s$ and $20s$. The number of the pipeline input data is 100.

	Global time (s)	Computation according to the frequency of getting final results (%)
Without FR	3105.442	97.249
FR: 3 workers	1181.837	87.998
FR: dynamic addition of workers	1409.444	–

Figure 19: Pipeline step parallelization using a farm construct. The pipeline is composed of 4 tasks. The execution time of each task is (in order): $10s$, $30s$, $5s$ and $5s$. Step 3 and 4 are collocated for load balancing. The number of the pipeline input data is 100. FR: Functional Replication

that the engine successively sends all inputs on component `T1`. These data are queued by the control part of the component. This latter is responsible to ensure one execution of task at a time and the order of treated input data (STCM specific behavior).

- `Pipe`: In this configuration, a pipeline skeleton construct is used to implement the *ForAll* control structure. The concrete assembly of the `Pipe` introduces two components: `distrib` and `collector` respectively responsible to split (collect) the *ForAll* input (output) data into several (one set of) data. All components are deployed when the control reaches the loop and deployed after retrieving all results

Figure 16 reports obtained performance results for the three configurations and for two different data set sizes: 10 and 100 doubles. The measures includes overheads related to the life cycle management of components. Many conclusions may be drawn. First, it is not conceivable to execute the loop in a sequential manner. Second, as expected, the `Pipe` configuration presents a more efficient execution. It is not sufficient to have a pipelined execution like in `Loop-Opt` and some life cycle management optimization to reach better performance but also an efficient implementation. Third, for longtime computations, the overhead of components life cycle management starts to become negligible. Finally, to achieve efficient execution and management it is necessary to consider the behavior of global composition, i.e. combined structures, in an assembly to decide a mapping on a concrete assembly.

5.4 Need of efficient behavioral skeleton management

A more advanced approach to deal with parallelism forms is to take benefits from behavioral skeleton constructs [3]. These constructs offer a powerful execution management of parallel applications. In particular, they consider not only efficient execution but also resources usage. This section presents two experiments aiming to illustrate principal advantages that behavioral skeletons should offer in the context of STKM. Both experiments are done for executing a pipeline composition with load-balanced stages. Load balancing is realized by either components collocation or by integrating functional replication skeleton. In this latter, we tested a fixed number of workers as well as dynamic addition of workers. The remainder of this section details each experiment and discusses obtained results.

The first experiment consists in testing the execution of a pipelined composition of non equivalent tasks, i.e with different computation times, and compare the execution performance with two distinct component placements. For the first placement, all components implicated in the pipeline are deployed on different machines. In the second case, the placement considers meta-data about computation duration of tasks. This meta-data is used to decide a possible collocation of components. The objective is to load balance the pipeline execution steps to optimize resources usage while preserving an efficient execution. That is a usual principle followed to management pipeline constructs. Collocation decisions in the context of the present work is done manually. Thus, for all cases, the concrete assembly is mapped to a pipeline skeleton construct as done in previous experiments. Figure 18 shows obtained results for a specific example. As expected, the results are close. Therefore, in addition to reach an efficient execution, it should be possible to improve resources usage.

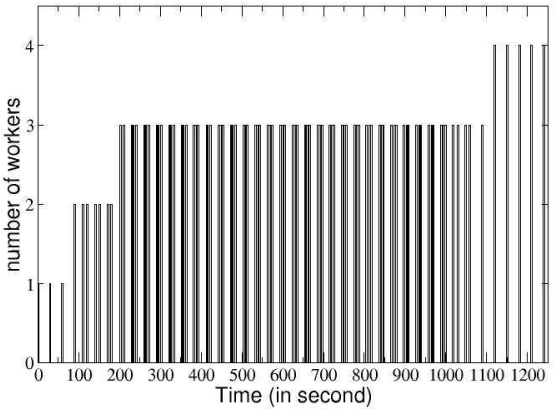

Figure 20: Dynamic management of workers in a behavioral farm skeleton construct. The farm construct is used to parallelize the second step of the pipeline construct of figure 19.

The second experiment uses another load-balancing approach. For a similar example but with different tasks, the principle is to parallelize the execution of a costly pipeline step. For that, this step is transformed to a functional replication construct in which the workers replicate the step. The structure of the functional replication is the one shown in Figure 6. Experiment details as well as the obtained results are shown in Figure 19. As can be noted, the execution time is divided by 3 in the second line. That corresponds to the maximum number of workers needed to achieve load balanced execution. That is in addition to last steps collocation. The last line corresponds to a test of dynamic worker addition by the manager of the behavioral skeleton. This manager implements a simple adaptation policy. This latter consists in the addition of a worker if the frequency of producing results on the output of the skeleton is more than a given value. In our particular case, this value is $10s$. Figure 20 shows the evolution of workers number during the test execution which results to the execution time reported in Figure 19. It is to note that the number of workers reaches 4. That is due to a limitation of the current implementation of the pipeline which assumes infinite buffers between tasks. In the scope of this report our objective is not to study pipeline or functional replication skeleton management issues. Performed experiments illustrates more the feasibility of realizing a behavioral skeleton in STKM and the benefits that such constructs should offer to STKM applications. Ongoing work should integrate already existing behavioral skeleton implementations.

6 Conclusions and future works

In this work we discussed experiments aimed at assessing the design of STKM. The experiments have been performed using SCA/Tuscany rather than GCM, the model originally extended by STCM that, in turn, evolved into STKM. This was due to the fact we were interested in investigating how GCM and STCM + STKM concepts in general were affected by the adoption of *state-of-the-art* technology, such as the one of Web Services. SCA represents a good compromise as it merges the SOA concepts with basic component features that allow to seamlessly migrate most of the CoreGRID/GCM experience into the service framework.

Experiments have been performed that i) measure the typical overheads involved in the usage of the SCA framework and ii) evaluate the performances achieved in those cases where STKM is supposed to support much better/performant implementations than plain STCM or skeletons.

The experiments evidenced that the overheads introduced when managing distributed applications on the SCA framework are relevant, and therefore the whole approach is only suitable and worth in case of coarse grain applications. This is not peculiar of SCA/Tuscany, however. Our direct experience in the GCM context demonstrated that similar results are achieved also in case of usage of other distributed application middlewares. In particular, within GridCOMP (`gridcomp.ercim.org`) we verified that using the GCM reference implementation built on top of

ProActive (`proactive.inria.fr`) overheads can be measured that are definitely very close to the ones experienced with SCA/Tuscany.

The experiments also evidenced that the optimizations introduced by STKM are worth, as they lead to better performances w.r.t. the very same applications implemented with usual STCM or GCM constructs and frameworks.

All the experiments were made hand coding the SCA/Tuscany source code that a STKM framework was supposed to generate. We have no actual complete implementation of STKM. By hand coding the STKM support we also verified the feasibility of implementing a full featured STKM programming environment on top of SCA/Tuscany. We verified that there are several limitation of the Tuscany prototype, not necessarily deriving from limitations in the SCA model, that impose the usage of different patches to support all the features needed by STKM. Being in contact with the Tuscany developers, we had the impression that some of the features required while implementing STKM and not yet in Tuscany will be available soon, due to their importance for different projects currently using Tuscany 1.2 implementation.

In the near future, we plan to concentrate on further improvements in the STKM design as well as in the design and implementation of a full version of STKM, possibly on top of SCA. In both cases the results of this work will be exploited.

References

[1] Marco Aldinucci, Hinde Bouziane, Marco Danelutto, and Christian Pérez. Towards software component assembly language enhanced with workflows and skeletons. In *Joint Workshop on Component-Based High Performance Computing and Component-Based Software Engineering and Software Architecture (CBHPC/COMPARCH 2008)*, 14-17 October 2008. To appear.

[2] Marco Aldinucci, Hinde Bouziane, Marco Danelutto, and Christian Pérez. Towards software component assembly language enhanced with workflows and skeletons. Technical Report 0153, CoreGRID Network of Excellence, 2008.

[3] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural Skeletons in GCM: Autonomic Management of Grid Components. In Didier El Baz, Julien Bourgeois, and Francois Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and Network-based Processing*, pages 54–63, Toulouse, France, February 2008. IEEE.

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

[5] Marco Aldinucci and Marco Danelutto. Securing Skeletal Systems with limited Performance Penalty: the Muskel Experience. *Journal of Systems Architecture*, 2008. In press. DOI: 10.1016/j.sysarc.2008.02.008.

[6] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Giorgio Zoppi. Advances in Autonomic Components & Services. In *Proceedings of the CoreGRID Symposium '80*, CoreGRID. Springer Verlag, August 2008. Las Palmas de Gran Canaria (E).

[7] 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, version 1.0. Technical report, Open Service Oriented Architecture collaboration (OSOA), March 2007.

[8] Carlo Bertolli, Massimo Coppola, and Corrado Zoccolo. The Co-replication Methodology and its Application to Structured 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, October 2007. ACM.

[9] Hinde Bouziane, Christian Pérez, and Thierry 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.

[10] Murray Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[11] Marco Danelutto and Giorgio Zoppi. Behavioural skeletons meeting services. In Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M.A. Sloot, editors, *Computational Science – ICCS 2008*, volume 5101 of *LNCS*, pages 146–153. Springer, 2008.

[12] Apache Software Foundation. Apache commons. `http://commons.apache.org/daemon/`.

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

[14] Tuscany home page, 2008. `http://tuscany.apache.org/`.