STKM on SCA: A Unified Framework with Components, Workflows and Algorithmic Skeletons

Marco Aldinucci[1], Hinde Lilia Bouziane[2], Marco Danelutto[3], and Christian Pérez[4]

[1] Dept. of Computer Science, University of Torino
aldinuc@di.unito.it
[2] LIP/Lyon 1 University, ENS Lyon
hinde.bouziane@ens-lyon.fr
[3] Dept. of Computer Science, University of Pisa
marcod@di.unipi.it
[4] LIP/INRIA, ENS Lyon
christian.perez@inria.fr

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

1 Introduction

Quite a large number of programming models have been and are currently proposed to support the design and development of large-scale distributed scientific applications. These models attempt to offer suitable means to deal with the increasing complexity of such applications as well as with the complexity of the execution resources (e.g. those in grids and/or clouds) while attempting to ensure efficient execution and resource usage. However, existing models offer peculiar features that actually make them suitable for specific kind of applications only. A current challenge is still to be able to offer a suitable programming model that easily and efficiently supports multi-paradigm applications.

Three different programming models have recently been considered to support large-scale distributed scientific applications: software components, workflows and algorithmic skeletons. All these models 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. *Software components* promote code reuse [1]. Components can be composed to build new and more complex components so that applications are component assemblies. Such assemblies are usually defined at compile time, although most component frameworks provide mechanisms that can be used to implement dynamic assemblies at run time. An assembly completely determines spatial interactions among components, and therefore it is referred to as *spatial* composition. Due to these features, component models are suitable to support strongly coupled compositions. *Workflow* models have been mainly developed to support composition of independent programs (usually loosely coupled and named tasks) by specifying temporal dependencies among them (and defining a *temporal composition*, in fact), to support efficient scheduling onto available resources (e.g. sites, processors, memories) [2]. Last but not least, *algorithmic skeletons* have been introduced to support typical parallel composition patterns (skeletons) [3]. Skeleton composition follows precise rules and skeleton applications are (well formed) skeleton assemblies. The composition style is mostly spatial, as in the software component case, although skeletons processing streams of input data sets usually present several embedded temporal composition aspects. Skeleton assembly "structures" can be exploited to provide automatic optimization/tuning for efficient execution on targeted resources.

In summary, each one of these three models is worth being used in some particular circumstances, but nevertheless all their properties seem to be relevant and worth to be considered in a single model. In [4], we discussed STKM (*Spatio-Temporal sKeleton Model*), a single programming framework providing programmers with components, workflows and skeletons. STKM allows these three abstractions to be mixed in arbitrary ways in order to implement complex applications. While in [4] we explored the *theoretical background* of STKM, in this paper, we concentrate on the problems related to the *implementation* of STKM components, realized as an extension of GCM (*Grid Component Model* [5]) components built on top of SCA (*Service Component Architecture* [6]).

This paper is organized as follows: Section 2 outlines relevant related work, Section 3 sketches the main features of STKM. SCA implementation design of STKM and related issues are presented in Section 4 while experimental results showing the feasibility of the whole approach are discussed in Section 5. Section 6 concludes the paper.

2 Background and Related Work

Skeleton based programming models allow application programmers to express parallelism by simply instantiating – and possibly nesting – items from a set of predefined patterns, the skeletons, that model common parallelism exploitation patterns [3]. Typical skeletons include both stream parallel and data parallel common patterns [7,8,9]. Programming frameworks based on algorithmic skeletons achieve a complete and useful separation of concerns between application programmers (in charge of recognizing parallelism exploitation patterns in the application at hand and of modeling them with suitable skeletons) and system programmers (in charge of solving, once and for all, during skeleton framework design and implementation, the problems related to the efficient implementation of skeletons and of skeleton composition). In turn, this separation of concerns supports rapid application development and tuning, allows programmers without specific knowledge on parallelism exploitation techniques to develop efficient parallel applications, and eventually supports seamless (from the application programmer perspective) porting of applications to new target architectures.

Skeleton technology has recently been adopted in the component based programming scenario by developing (composite) components modeling common parallelism exploitation patterns and accepting other components as parameters to model the skeleton inner computations [10,11]. To support automatic adaptation of skeleton component execution to highly dynamic features of target architectures such as grids, autonomic management has been eventually combined with skeleton based modeling in the *behavioural skeletons* [12]. The result is that behavioural skeletons – or a proper nesting of behavioural skeletons – can be simply instantiated to obtain fully functional, efficient parallel applications with full, autonomic auto tuning of performance concerns.

Component based skeleton frameworks provide all the optimizations typical of algorithmic skeletons. Behavioural skeletons add the autonomic tuning of non-functional concerns. However, no support has been provided, as far as we know, to support skeleton components in workflows. Skeletons in workflows would allow to express computations as temporal compositions while preserving the possibility to optimize parallel workflow stages according to the well-known results of the skeleton technology.

3 STKM

STKM (*Spatio-Temporal sKeleton Model* [4]) extends STCM [13], a *Spatio-Temporal Component Model* merging component and workflow concepts, with (behavioural) skeleton support. This extension promotes more simplicity of design and separation of functional concerns from non-functional ones (management of components life cycle, parallelism, etc.). It also promotes the portability of applications to different execution contexts (resources). For that, a level of abstraction is offered to allow a designer to express the functional behaviour of an application through its assembly. The behaviour expressiveness is exploited by an STKM framework to adapt the application to its execution context. This section outlines the unit of composition of STKM, its assembly model and a suitable approach to manage the assembly by the framework.

An STKM *component* is a combination of a classical software component and task (from workflows) concepts. As shown in Figure 1, a component can define *spatial* and/or *temporal* ports. Spatial ports are classical component ports. They express interactions between components concurrently active [13]. Temporal ports (input/output) behave like in a workflow, instead. They express data dependences between component tasks and then an execution order of these tasks. Like in a workflow, components in-

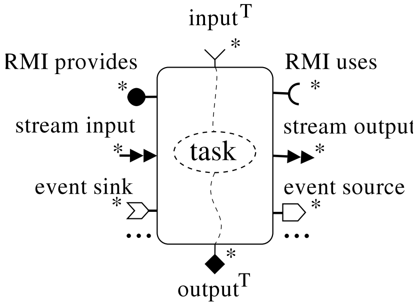


Fig. 1. An STKM component. T: refers to temporal ports. Other ones are spatial.

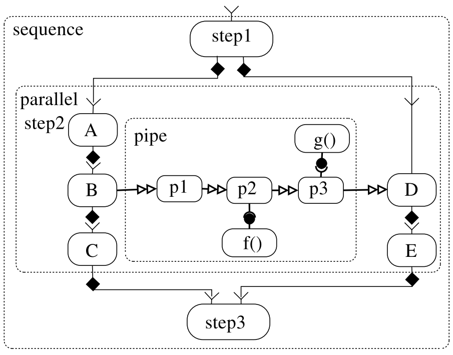


Fig. 2. Example of an STKM assembly

volved in a temporal relation can be instantiated when both control and data flows reach them [13]. The difference is that the life cycle of an STKM component may be longer than the completion of a task in a workflow.

An STKM *assembly* describes spatial and temporal dependencies between components. Spatial dependencies are drawn by spatial port connections. Temporal dependencies are drawn by a data flow (input and output port connections) and a control flow (using constructs like sequences, branches (*if* and *switch*), loops (*for* and *while*), etc.). In addition, STKM defines constructs dedicated to skeleton-based parallel paradigms. These constructs are particular composite components representing skeleton schemes (pipe, farm, functional replication, etc.). They can be composed with components and/or other skeletons at different levels of an assembly. This ability preserves the pragmatic of skeletons. Figure 2 shows a graphical STKM assembly example of a sequence `(step1; step2; step3)`, where `step2` is a parallel composition of sequences `(A;B;C)` and `(D;E)`. A spatial dependency is drawn between these sequences, as component `D` consumes the results of the `(pipe)` skeleton construct whose inputs are produced by `B`. The figure also shows that skeleton elements (the pipe stages) may express dependences with other components (those providing f and g). These further components belong to the same assembly of the pipeline skeleton. In general, skeleton assemblies follow a fixed schema, the one typical of the parallel pattern implemented by the skeleton. In this case, the assembly is a linear assembly of stages. Each stage, may involve any number of components in a sub-assembly, provided that it has an input (output) stream port to be connected to the previous (next) pipeline stage.

To be executed, an STKM application needs to be transformed into a concrete assembly, that may include a specific engine. This engine represents *de facto* STKM run time. This run time comprehends the mechanisms needed to support both skeleton and workflow aspects of STKM components. Therefore, it differs in the way STKM components are processed to produce the concrete component assembly from other systems that only concentrate on skeletons [14] or worflows [15]. The concrete assembly may introduce non-functional concerns like data flow management or the hidden part of skeletons implementations (cf. Section 4.1). The STKM engine manages the life cycle of components and the execution order of tasks. During the transformation, STKM benefits from the expressive power of an assembly to exploit the maximum parallelism from a part or from the whole assembly, and to adopt an adequate scheduling policy depending on actually available execution resources. Thus, both explicit (using skeleton constructs) and implicit parallelisms can be considered. An example for the last case is the mapping of a `forAll` loop to a functional replication skeleton in which the workers are the body of the loop. Such a mapping should allow STKM to take benefits from already existing (behavioural) skeleton management mechanisms able to deal with performance [16] concerns, security [17], fault tolerance [18], etc.

4 An SCA Based Implementation of STKM

This paper aims at evaluating the feasibility of the STKM model. SCA (*Service Component Architecture* [6]) is used to investigate whether porting STKM concepts to the Web Service world is as effective as porting other GCM concepts, as shown in [19]. As

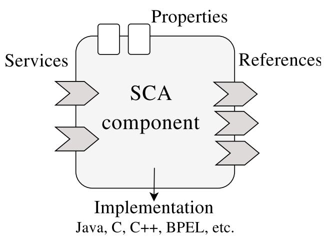


Fig. 3. An SCA component

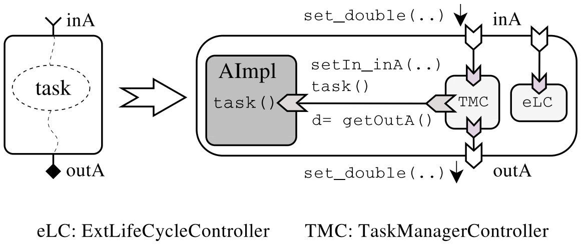


Fig. 4. From an STKM component to an SCA one

STCM was devised as an extension of GCM [13], it was natural to explore the feasibility of STKM in the same direction.

SCA is a specification for programming applications according to a *Service Oriented Architecture* (*SOA*). It aims to enable composition of services independently from the technologies used to implement them and from any SCA compliant platform. SCA specifies several aspects: assembly, client and component implementation, packaging and deployment. The *assembly* model defines a component as a set of ports named *services* and *references* (Figure 3). Ports can be of several kinds such as Java, OMG IDL (*Interface Definition Language*), WSDL (*Web Services Description Language*), etc. They allow message passing, Web Services or RPC/RMI based communications. The interoperability between communicating components is ensured through dedicated *binding* mechanisms. The assembly of components may be hierarchical. The hierarchy is abstract, thus allowing to preserve encapsulation and simplifying assembly process. The *client and implementation* model describes a service implementation for a set of programming languages. Several means are used: annotations for Java/C++ or XML extensions for BPEL [20], etc. These means permit the definition of services, properties, and meta-data like local or remote access constraints associated to a service. Last, the *packaging and deployment* model describes the unit of deployment associated to a component. For deployment concerns, an SCA platform is free to define its own model.

4.1 Mapping STKM Concepts on SCA

Two main issues arise when implementing STKM on SCA: the projection of the user view of an STKM component to an SCA based implementation, and the management of an STKM assembly at execution. This section discusses these two issues.

STKM Component and Ports. As mentioned earlier, an SCA based implementation of GCM components has been already proposed [19]. It is based on mapping components, client/server ports, controllers and implementations on a set of services/references, SCA components and SCA implementations. This paper does not detail this projection. It focuses on describing a projection of the concepts added by STKM, i.e. temporal ports (inherited from STCM) and skeleton constructs.

An STKM component is mapped to an SCA composite component and temporal ports to a set of services/references and controllers as illustrated in Figure 4. A component task appears as a service provided by the user implementation. Temporal ports are mapped to a set of services provided/used by a transparent control part of the component. This control part (components TMC and eLC in Figure 4) is responsible to manage input and output data availability and task executions. Details about this management can be found in [13]. Note that realizing controllers using components is not a new idea. This promotes composability and code reuse.

STKM skeleton constructs, which are also components, are projected with a similar approach. The difference is that an implementation of a skeleton may consider additional non-functional elements like managers for behavioural skeletons. A simplified example is shown in Figure 5. The figure represents a functional replication behavioural skeleton. Components MGR (manager), E (emitter) and C (collectors) belong to the non-functional part of the skeleton.

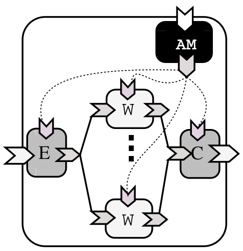
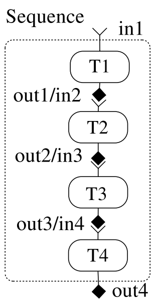


Fig. 5. Functional replication behavioural skeleton implementation, without the control part shown in Figure 4.

Thus, a simple projection of STKM components to SCA components is possible thanks to the full support of RPC/RMI ports and hierarchical composition in SCA.

STKM Assembly. Not only STKM makes an assembly more explicit with respect to the behaviour of an application, but it also aims to enable automatic and dynamic modifications of the assembly. Thus, it is not sufficient to have just a projection of components as described above. It is also necessary to introduce mechanisms to deal with data transfer between dependent components, to order task execution according to both data flow and control flow, and to manage the life cycle of components. Several solutions can be proposed. This paper presents a simple solution, based on a distributed data transfer and a centralized orchestration engine.

With respect to *data management*, the data transfer between data flow connected components can be direct if both components simultaneously exist or it can be indirect through a proxy component, in case the target component does not exist yet. Figure 7 illustrates how such a proxy can be introduced between two components of the sequence



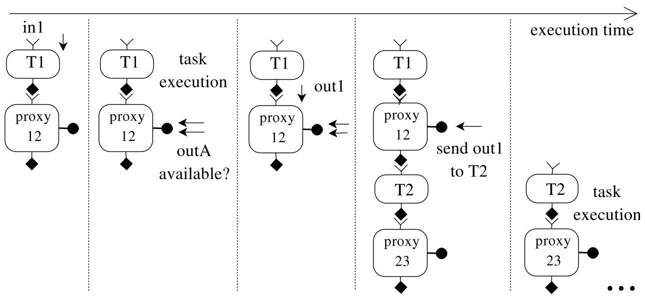


Fig. 6. An STKM sequence example (user view)

Fig. 7. Data transfer through proxy components and dynamic changes of the sequence assembly. The STKM, rather than SCA, notation for ports is still used for simplicity.

assembly of Figure 6. During the execution, `proxy12` receives the output of `T1` and waits a request to send it to `T2` (from the STKM engine, see paragraph below). The availability of the proxy is assumed to be at the responsibility of the transformation engine, which is out of the scope of this paper.

The STKM *engine* is an SCA client program. It contains the sequence of actions which create/destroy components, connect/disconnect ports, manage data availability/transfer and the order of tasks executions. These actions are deduced from the behaviour expressed by the STKM assembly. Figure 7 shows a part of the dynamic assembly evolution of Figure 6 managed by a naive STKM engine which creates components when the data/control flow reaches them. Details about this engine can be found in [21].

4.2 Usage of Tuscany Java SCA

We base the proposed SCA implementation on Tuscany Java SCA Version 1.2.1 [22]. Tuscany is still under development but it provides a preliminary support for distributed executions. However, some features affecting the support of some STKM requirements are not yet supported. The more relevant missing features are related to the dynamicity of an assembly. In Tuscany 1.2.1, to dynamically add/remove components, there are mainly two approaches. The first one is based on dynamic reconfiguration of `contributions` (components packages). However, it requires to stop the application execution[1]. The second approach is based on the addition/removal of *nodes*. A *node* is a process hosting one or more component instances on a given execution resource. The approach does not require to suspend an application execution. However, Tuscany requires nodes to be statically defined. We select this approach and make all decisions static.

SCA specification does not offer an API to connect/disconnect references to services. However, it provides an API to allow passing service references. Using this API, we specified connection/disconnection operations. These operations are associated to a port and exposed as a service implemented by the non-functional part of a component. Note that the service passed by reference is accessed using a Web Service binding protocol in the used Tuscany version.

The advantage of our approach to overcome Tuscany lacking or non-properly supported features, is that no modification in the distribution was needed. However, as discussed in next section, some experiment results are affected by the adopted solutions. For the purpose of this paper, however, this does not prevent the possibility to demonstrate the benefits of STKM and its feasibility on a Web Service based environment.

5 Evaluation

To evaluate the benefits of STKM, this section discusses the performance of the proposed SCA based implementation. We developed an application according to different compositions: sequence, loop, pipeline and nested composition of pipeline and farm/functional replication skeleton constructs, such that we could experiment various

[1] Efforts have been made to overcome this limitation [23]. Its feasibility was proved using Tuscany 1.1.

	Time in s
Remote node Launching	45.56
Programmed port connection	3.20

Fig. 8. Average of times to deploy and connect components

RTT in ms	Intra-node Inter-comp.	Inter-Node Intra-host	Inter-Node Inter-host
Default protocol	0.076	20.35	20.17
WS protocol	22.66	24.23	24.11

Fig. 9. Round Trip Time (RTT) in ms on a local Ethernet network for different situations. WS: Web Service

execution scenarios for a same application with respect to different execution contexts. The application is synthetic and its parameters (amount of involved data, relative weight of computation steps, etc.) have been tuned to stress the different features discussed.

The projection from STKM assembly to SCA components and an STKM engine was manually done using Tuscany Java SCA version 1.2.1 and Java 1.5. The deployment of components was done manually by using `scp` and their creation was done at the initiative of the engine by using `ssh`. Experiments have been done on a cluster of 24 Intel Pentium 3 (800 MHz, 1 GB RAM), running Linux 2.4.18 and connected by a 100 MBit/s switched Ethernet.

5.1 Metrics

We first evaluated basic metrics, including overheads of dynamic life cycle management – nodes launching and port connections – and inter component communications. The results are displayed in Figure 8 and Figure 9. First, it must be pointed out that starting a node is an expensive operation. This is due to remote disk accesses, done through NFS, required to launch a JVM and to load a node process by the used common-daemon library[2]. The time is also stressed by the significant usage of *reflection* and dynamic class loading by Tuscany as well message exchanges between a node and *an* SCA *domain* (application administrator) for services registry/checking. Second, the time necessary to connect ports through the reference passing mechanism (Section 4.2) is explained by the serialization/deserialization activity needed to pass a reference, and by multiple *reflection* calls. Globally, the overheads stem from the framework and from "tricks" explained in Section 4.2. The round trip times are measured for an empty service invocation for two binding protocols (default "unspecified" SCA and WS) and various placements of components. As explained in Section 4.2, the WS protocol is used for services passed by reference. The results show that the choice of a binding protocol affects communication times. There is a quite important software overhead, that makes network overhead almost negligible in next experiments. This is not a surprise as SCA specification addresses this issue and claims that SCA is more adequate for coarse grain codes or wide area networks.

5.2 Benefits of Resources Adaptation Capability: A Sequence Use Case

This section studies the impact on performance when mapping an STKM assembly to different concrete assemblies. A sequence of 4 tasks (Figure 6) is mapped to the configurations presented in Figure 10. This leads to two concrete assemblies for the abstract

[2] Apache commons: `http://commons.apache.org/daemon`

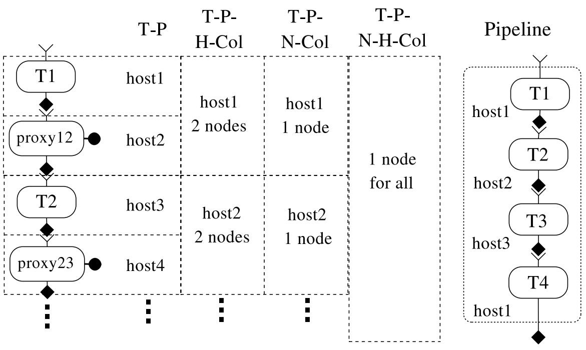


Fig. 10. Configurations used in Figure 11. T: Task in component, P: Proxy, H: Host, N: Tuscany Node, Col: Collocation

	Efficiency (%)
T-P	27.03
T-P-H-Col	20.13
T-P-N-Col	28.38
T-P-N-H-Col	28.50
Pipeline	51.14

Fig. 11. Efficiency to execute a sequence of 4 tasks. The execution time of each task is $20s$. The life cycle of a component is delimited by input/output data availability

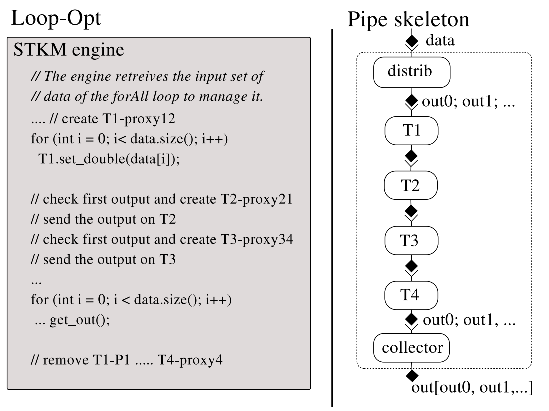
one shown in Figure 6 and a given placement of components. For the assembly on the left part of Figure 10, the execution and life cycle management of components follows the principle presented in Section 4.1. On the right part, the sequence is mapped to a pipeline composition. In this case, all concerned components are deployed on different hosts and connected by the engine *before* the execution of the first task.

Figure 11 reports the execution efficiency in terms of the percentage of the whole execution time (including remote (Tuscany) node creation, component instantiation and port connections) spent in the sequence computation. Without surprise, the `Pipeline` configuration leads to a more efficient execution. This will remain true even if the overheads are lower for coarser grain codes. However, the performance criterion may be not sufficient to choose a particular configuration. Other criteria, such as resource usage, could also be taken into account. Therefore, the `Pipeline` configuration may cause an over-consumption of resources. That may be problematic when using shared resources, like Grids. Other configurations may offer the ability to optimize resource usage with efficient scheduling policies. The objective of this paper is not to study such policies, so they are not treated and only base cases are tested.

The `T-P-N-H-Col` configuration behaves better because all components are executed in the same process. Also, the lazy instantiation of components in Tuscany reduces the number of concurrent threads. However, components in a sequence may require different nodes because of execution constraints on memory or processor speed. In such a case, the `T-P-N-Col` configuration may be more suitable. For the remainder of this section, we selected this configuration.

5.3 Need for Form Recognition

STKM promotes the ability to recognize implicit parallelism forms from an assembly and to exploit them to improve the performance when possible. For that, the approach is to map recognized parallelism forms to a composition of skeleton constructs. This section illustrates the benefits of this approach through a sequential *for* loop. The body of this loop is the sequence shown in Figure 6. The sequence is assumed to be stateless



```
// The engine retreives the input set of
// data of the forAll loop to manage it.
.... // create T1-proxy12
for (int i = 0; i< data.size(); i++)
  T1.set_double(data[i]);

// check first output and create T2-proxy21
// send the output on T2
// check first output and create T3-proxy34
// send the output on T3
...
for (int i = 0; i < data.size(); i++)
 ... get_out();

// remove T1-P1 ..... T4-proxy4
```

Fig. 12. Overview of the configurations used for executing the *For* loop in a parallel way

	Efficiency (%)
data size: 10x	
Loop	28.40
Loop-Opt	56.16
Pipe	76.96
data size: 100x	
Loop	28.40
Loop-Opt	90.68
Pipe	95.90

Fig. 13. Results of executing a loop according to different configurations. The loop body is a stateless sequence of 4 tasks of $20s$

and to be executed for each element of an array of data (input of the loop). The output of the loop is the collection of all the outputs of the different iterations. To execute such a loop, let us study three configurations: *Loop*, *Loop-Opt* and *Pipe*.

In the *Loop* configuration, no parallelism form is recognized. All iterations are executed sequentially and the sequence is mapped to the `T-P-Node-Col` configuration shown in Figure 10. The *Loop-Opt* configuration exploits the independence of the iterations and the fact that components are stateless to make use of a pipelined execution. The concrete assembly is the same as for the `Loop` configuration. The difference occurs in the management of the data flow and the life cycle of components as shown in Figure 12. As can be noted, the STKM engine is responsible to split/collect the loop input/output data. The pipelined execution is done thanks to the control part of components. This part is responsible to queue received data and to ensure one task execution at a time and the order of treated data (STCM specific behaviour). Note that a component $\mathtt{T}_i$ is instantiated once the first output of $\mathtt{T}_{i-1}$ is available and removed after the loop execution. The *Pipe* configuration projects the loop to a pipeline skeleton construct. The result is shown on the right part of Figure 12. It introduces two components: `distrib` and `collector` responsible to respectively split (collect) the loop input (output) data into several (one set of) data. All components are instantiated when the control flow reaches the loop and destroyed after retrieving the loop result.

Figure 13 reports results for two different data set sizes. The measures include the overhead related to the life cycle management. Several conclusions can be drawn. First, the `Pipe` configuration presents a more efficient execution. In fact, it is not sufficient to have a pipelined execution (`Loop-Opt`) to reach better performance: an *efficient* implementation of a pipeline, such as one of those available exploiting assessed skeleton technology, is also needed. Second, the overhead of life cycle management can as usual be hidden with longer computation time. Third, in order to achieve efficient execution and management it is necessary to consider the behaviour of the global composition, i.e. combined structures, in an assembly when mapping on a concrete assembly.

	Global time (in s)
Without FR[a]	3105
Farm: 3 workers	1182
FR: dynamic addition of workers	1409

Fig. 14. 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.

[a] FR: Functional Replication

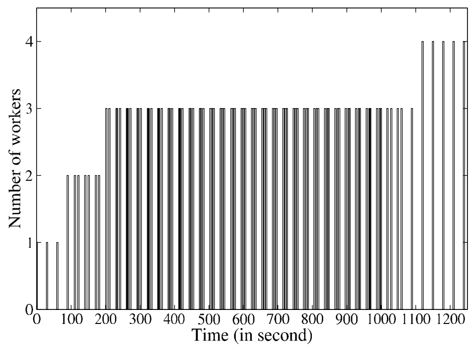


Fig. 15. Dynamic management of workers in a behavioural farm skeleton construct. The farm construct is used to parallelize the second step of the pipeline construct of figure 14.

5.4 Need for Efficient Behavioural Skeleton Management

This section presents experiments illustrating the advantage that behavioural skeletons should offer in the context of STKM. Experiments are relative to the execution of a pipeline composition. The mapping of this composition on a concrete assembly is directed by two criteria: performance and resource usage. For a pipeline, improving these criteria is usually achieved by load-balancing its stages, collocating stages and/or parallelizing bottleneck stages (e.g. by integrating a functional replication skeleton).

The *stage collocation* experiment compares the execution of 4 pipelined tasks with and without applying load balancing. The computation times of the tasks are, in order: $10s$, $15s$, $5s$ and $20s$. To load balance the stages, the approach was to collocate the second and third stages to be executed on a same host. The efficiency is 96.04% for the collocation scenario and 95.92% without collocation. Thus, it is possible to improve resource usage while preserving performance. While the load balancing was done manually for this experiments, a behavioural skeleton implementation is expected to do it automatically, by monitoring the task execution times and then adequately "grouping" pipeline stages to match the pipeline ideal performance model.

The *stage parallelization* experiment compares the execution of a pipeline composition according to two load-balancing approaches. The pipeline has 4 stages of durations $10s$, $30s$, $5s$ and $5s$. The objective is to parallelize the execution of a costly pipeline step, here $30s$. In addition to the possibility to collocate the third and fourth stages, the second stage is mapped a) to a farm skeleton (Figure 5) with a static number of workers – three in this case – or b) to a "replication" behavioural skeleton (Figure 5) with autonomic and dynamic addition of workers [12]. The skeleton manager implements a simple adaptation policy which adds a worker if the output frequency of the skeleton is less than a given value, $10s$, in this case. The results are shown in Figure 14 and Figure 15. As expected, the whole execution time is improved – by a 2.6 factor – when using a farm with static number of workers. For dynamic workers addition, the number of workers reaches 4 with an improvement of 2.2 instead of 2.6. This is due to the adaptation phase overhead and to the fact that adaptation does not rebalance tasks already in the workers queues. The experiment illustrates the feasibility of realizing a behavioural skeleton in STKM as well as the ability to preserve the advantages of such constructs.

6 Conclusion and Future Works

STKM is a model that aims to increase the abstraction level of applications with enabling efficient executions. Its originality is to unify within a coherent model three distinct programming concepts: components, workflows and skeletons. This paper evaluates its feasibility and its benefits. The proposed mapping introduces a set of non-functional concerns needed to manage an STKM assembly; concerns that can be hidden to the end user and that can be used for execution optimizations. Hand-coded experiments show that STKM can lead to both better performance and resource usage than a model only based on workflows or skeletons.

Future works are fourfold. First, model driven engineering techniques should be considered for implementing STKM to automatize the assembly generation and existing component based behavioural skeleton implementations [19]. Second, investigations have to be made to understand which assembly has to be generated with respect to the resources and to criteria to optimize. Third, techniques to optimize an application are also needed. Fourth, a validation on real word applications [4] has to be done. For that, it should be worth investigating the usage of recent SCA implementation overcoming part of the limitations of the one we used here (e.g. the recently available Tuscany version 1.4).

References

1. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley/ACM Press (2002)
2. Fox, G.C., Gannon, D.: Special issue: Workflow in grid systems: Editorials. Concurr. Comput.: Pract. Exper. 18(10), 1009–1019 (2006)
3. Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing 30(3), 389–406 (2004)
4. Aldinucci, M., Danelutto, M., Bouziane, H.L., Pérez, C.: Towards software component assembly language enhanced with workflows and skeletons. In: Proc. of the ACM SIGPLAN compFrame/HPC-GECO workshop on Component Based High Performance, pp. 1–11. ACM, New York (2008)
5. CoreGRID NoE deliverable series, Institute on Programming Model: Deliverable D.PM.04 – Basic Features of the Grid Component Model, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf` (assessed, February 2007)
6. Beisiegel, M., et al.: SCA Service Component Architecture - Assembly Model Specification, version 1.0. TR, Open Service Oriented Architecture collaboration (OSOA) (March 2007)
7. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)

8. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
9. Caromel, D., Leyton, M.: Fine Tuning Algorithmic Skeletons. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 72–81. Springer, Heidelberg (2007)
10. Aldinucci, M., Campa, S., Coppola, M., Danelutto, M., Laforenza, D., Puppin, D., Scarponi, L., Vanneschi, M., Zoccolo, C.: Components for high performance Grid programming in Grid.it. In: Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications, Saint-Malo, France. CoreGRID series, pp. 19–38. Springer, Heidelberg (2005)
11. Gorlatch, S., Duennweber, J.: From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In: Future Generation Grids. Springer, Heidelberg (2005); selected works from Dagstuhl 2005 FGG workshop 2005
12. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Dazzi, P., Laforenza, D., Tonellotto, N., Kilpatrick, P.: Behavioural skeletons in GCM: autonomic management of grid components. In: Baz, D.E., Bourgeois, J., Spies, F. (eds.) Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing, Toulouse, France, pp. 54–63. IEEE, Los Alamitos (2008)
13. Bouziane, H.L., Pérez, C., Priol, T.: A software component model with spatial and temporal compositions for grid infrastructures. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 698–708. Springer, Heidelberg (2008)
14. Falcou, J., Sérot, J.: Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library. In: Parallel Computing: Architectures, Algorithms and Applications. NIC, vol. 38, pp. 243–252. John Von Neumann Institute for Computing, Julich (2007)
15. Yu, J., Buyya, R.: A Taxonomy of Workflow Management Systems for Grid Computing. Journal of Grid Computing 3(3-4), 171–200 (2005)
16. Aldinucci, M., Danelutto, M.: Algorithmic Skeletons Meeting Grids. Parallel Computing 32(7), 449–462 (2006)
17. Aldinucci, M., Danelutto, M.: Securing skeletal systems with limited performance penalty: the Muskel experience. Journal of Systems Architecture 54(9), 868–876 (2008)
18. Bertolli, C., Coppola, M., Zoccolo, C.: The Co-replication Methodology and its Application to Structured Parallel Programs. In: CompFrame 2007: Proc. of the 2007 symposium on Component and framework technology in high-performance and scientific computing, pp. 39–48. ACM Press, New York (2007)
19. Danelutto, M., Zoppi, G.: Behavioural skeletons meeting services. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part I. LNCS, vol. 5101, pp. 146–153. Springer, Heidelberg (2008)
20. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0 (oasis standard). Technical report (2006)
21. Aldinucci, M., Bouziane, H., Danelutto, M., Pérez, C.: Towards a Spatio-Temporal sKeleton Model Implementation on top of SCA. Technical Report 0171, CoreGRID Network of Excellence (2008)
22. Apache Software Found: Tuscany home page, WEB (2008), `http://tuscany.apache.org/`
23. Aldinucci, M., Danelutto, M., Zoppi, G., Kilpatrick, P.: Advances in autonomic components & services. In: Priol, T., Vanneschi, M. (eds.) From Grids To Service and Pervasive Computing (Proc. of the CoreGRID Symposium 2008), CoreGRID, Las Palmas, Spain, pp. 3–18. Springer, Heidelberg (2008)