Certification of Workflows in a Component-Based Cloud of High Performance Computing Services

A. B. de Oliveira Dantas[1], F. H. de Carvalho Junior[1], and L. S. Barbosa[2]

[1] MDCC, Universidade Federal do Ceará
Campus do Pici, Fortaleza, Brazil
{allberson,heron}@lia.ufc.br
[2] HASLab INESC TEC & Universidade do Minho
Campus de Gualtar, Braga, Portugal
lsb@di.uminho.pt

Abstract. The orchestration of high performance computing (HPC) services to build scientific applications is based on complex workflows. A challenging task consists of improving the reliability of such workflows, avoiding faulty behaviors that can lead to bad consequences in practice. This paper introduces a certifier component for certifying scientific workflows in a certification framework proposed for HPC Shelf, a cloud-based platform for HPC in which different kinds of users can design, deploy and execute scientific applications. This component is able to inspect the workflow description of a parallel computing system of HPC Shelf and check its consistency with respect to a number of safety and liveness properties specified by application designers and component developers.

1 Introduction

Contrariwise to other engineering disciplines, reliability is often disregarded in current software development. Actually, testing and *a posteriori* empirical error detection dominate the practice of software industry, compared to formal verification and correct-by-construction techniques.

The problem is that software and computational systems are inherently complex. Each line of code is a potential source of errors, and programs often exhibit a huge number of potential states, making it difficult to predict their behavior and verify their properties in a rigorous way. This difficulty is more evident in emerging heterogeneous computing environments in High Performance Computing (HPC), where concurrent programs are omnipresent.

Scientific Workflow Management Systems (SWfMS) have been largely applied by scientists and engineers for the design, execution and monitoring of reusable data processing task pipelines in scientific discovery and data analysis [24]. In these systems, workflows are commonly represented by components that absorb all the orchestration logic required to solve a specific problem. Applications emerge by composition of workflows and different sorts of computational components, usually provided in generic or tailored libraries. Increasing the reliability of the available workflows is considered a challenging task in the sense that

deadlocks must be avoided, crucial operations must be effectively executed, and no faulty behaviors should be induced as a result of badly designed workflows.

HPC Shelf is a proposal of a component-oriented platform to provide cloud-based HPC services. It offers an environment to develop applications matching the needs of specialists (i.e. experts on the relevant scientific or engineering domain) who are supposed to deal with domain-specific, heavy computational problems by orchestrating a set of parallel components tuned to classes of parallel computing platforms. Orchestrations in HPC Shelf are driven by a SWfMS called SAFe (*Shelf Application Framework*) [9]. Parallel computing systems implement applications in HPC Shelf by composing components that address functional and non-functional concerns, representing both hardware and software elements.

In such a scenario, this paper approaches the problem of *certifying* scientific workflows through the verification of typical behavioral properties (e. g. safety and liveness) they are expected to exhibit, therefore increasing their confidence levels. We are interested not only in discovering design errors on workflows, but also on improving their specifications based on the verification results. This work is based on a certification framework [10] previously proposed by the authors for HPC Shelf. Such a framework is basically a VaaS (Verification-as-a-Service) platform, where *certifier* components can be created and connected to other (certifiable) components within a parallel computing system under design. Certifier components orchestrate a set of *tactical* components in certification tasks. The latter, on the other hand, encapsulate the functionalities of one or more existing verification infrastructures, commonly composed of provers, model checkers and other elements, and run on parallel computing platforms of HPC Shelf, for accelerating the certification process.

This paper proposes a new kind of certifier components, designated by SWC2 (Scientific Workflow Component Certifier), for statically certifying workflows of parallel computing systems over the parallel computing infrastructure of HPC Shelf. Some of the main patterns found in this class of workflows and their verification are discussed. The proposed approach is further illustrated by resorting to a specific SWC2 component and a related tactical component, which encapsulates the mCRL2 verification toolset.

Related work. Several studies on the formalization and verification of business workflows have been proposed in literature. These include Event-Condition-Action rules (triggers) [6, 14, 18], logic-based methods [3, 5, 23], Petri Nets [2, 1, 25] and State Charts [27]. The approach proposed here, however, is innovative in the sense that no initiatives were found regarding the formalization and verification of scientific workflow patterns.

Paper structure. HPC Shelf is succinctly presented in Section 2. Section 3 presents an overview of the certification framework. Section 4 introduces workflow certifiers. Section 5, in turn, discusses the way workflows in HPC Shelf are translated to behavioral models in mCRL2. The approach is illustrated with a case study in Section 6. Finally, Section 7 concludes the paper.

2 HPC Shelf

HPC Shelf is a cloud computing platform for cloud-based HPC applications. It receives problem specifications from *specialist users* and build computational solutions for them. For that, the platform offers, to *application providers*, tools for building parallel computing systems by orchestrating components that comply to Hash, a model of intrinsically parallel components [7], representing both computations (software) developed by *component developers*, and parallel computing platforms (hardware) offered by *platform maintainers*. Specialists, providers, developers and maintainers are the stakeholders of HPC Shelf.

2.1 Component Kinds in HPC Shelf

HPC Shelf defines different kinds of components: virtual ***platforms***, representing distributed-memory parallel computing platforms; ***computations***, implementing parallel algorithms by exploiting the features of a class of virtual platforms; ***data sources***, storing data that may interest to computations; ***connectors***, which couple a set of computations and data sources placed in distinct virtual platforms; and ***bindings***, for connecting *service* and *action* ports exported by components for communication and synchronization of tasks, respectively.

A *service binding* connects a *user* to a *provider* port, allowing a component to consume a service offered by another component. In turn, *action bindings* connect a set of action ports that export the same set of action names. Two actions of the same name whose action ports are connected in two components execute when both components make them active at the same time (*rendezvous*). Figure 1 depicts a scenario illustrating components and their bindings.

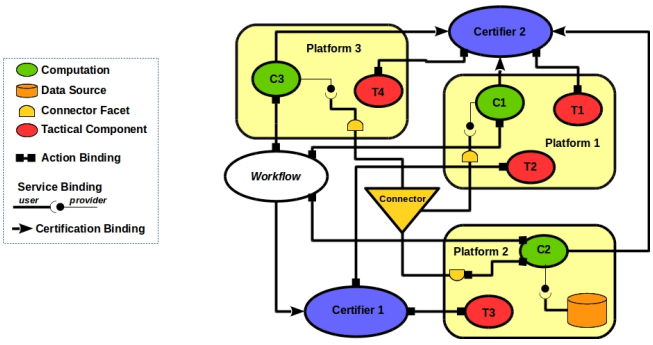

Fig. 1: Components in a hypothetical parallel computing system.

Components have a predefined action port, called *LifeCycle* which respond to a number of actions for life-cycle control. Action **resolve** selects a component implementation and a virtual platform for it, according to a system of *contextual*

contracts (see below). Action **deploy** deploys a selected component in a parallel computing platform. Action **instantiate** instantiates a deployed component, which becomes ready for communication with other components through service and action ports. Finally, action **release** releases a component from the platform on which it is instantiated, when it is no longer useful.

2.2 Architecture

The architecture of HPC Shelf is structured around three main elements, Front-End, Core and Back-End, as described below.

The Front-End is SAFe (*Shelf Application Framework*) [9], a collection of Java or C# classes and design patterns that *providers* use for deriving applications. They use *SAFe Scientific Workflow Language* (SAFeSWL) for specifying parallel computing systems, divided into an *architectural* and an *orchestration* subsets. The former is used to specify solution components and their bindings. The later orchestrates them. The workflow component of a parallel computing system is a special connector that runs in SAFe for performing the SAFeSWL orchestration.

The Core manages the life-cycle of components, offering services for *developers* and *maintainers* register components and their contracts. For that, the Core implements an underlying component resolution mechanism based on *contextual contracts*. Applications access the services of the Core for resolving contracts and deploying the components of their parallel computing systems.

The Back-End is a service offered by a *maintainer* to the Core for the deployment of virtual platforms. Once deployed, virtual platforms may communicate directly with the Core for instantiating components, which become ready for direct communication with applications through service and action bindings.

2.3 Contextual Contracts

HPC Shelf employs a system of *contextual contracts* [8] that separates interface and implementation of components (abstract and concrete components, respectively), so that one or more concrete components may exist in the Core's catalog for a given abstract component. Different components will meet different assumptions on the requirements of the host application and the features of the parallel computing platforms where they can be instantiated (*execution context*). For that, an abstract component has a *contextual signature*, composed of a set of *context parameters*. In turn, a concrete component must be associated with a type, i.e. a *contextual contract*, defined by an abstract component and a set of *context arguments* that valuate its context parameters. During orchestration, when the action **resolve** of a component is activated, a resolution procedure is triggered to choose a concrete component matching the corresponding contract.

3 The Certification Framework

We have proposed a certification framework for components in HPC Shelf [10], which introduced the kind of ***certifier*** components. The connection between

a certifier and a certifiable component is called *certification binding*. It also introduced the auxiliary kind of ***tactical*** components, for encapsulating proof infrastructures orchestrated by certifiers in verification tasks.

3.1 Parallel Certification Systems

A parallel certification system is like a parallel computing system for performing the certification procedure of a certifier component. It comprises a set of tactical components, deployed in virtual platforms where the required verification infrastructures are deployed; a certifier component, which orchestrates the tactical components through a workflow written in TCOL (Tactical Component Orchestration Language); a set of certifiable components, linked to the certifier through certification bindings. The certifier runs on SAFe, like the workflow component.

For a certifiable component to be certified, one or more certifier components whose certification ports are compatible to the certification port of the certifiable component must be chosen from the catalog and connected to it through certification bindings. When the workflow activates the new action **certify** in the life-cycle port of the certifiable component, the parallel certification system of each certifier performs its certification procedure to certify it.

The certification of a component with respect to a certifier component is idempotent, that is, it is executed once, even though **certify** is activated multiple times in one or more applications. Each certifier distinguish which properties it may verify are either mandatory or optional. At the end of the certification process, the component is considered to be certified with respect to the certifier component if all mandatory properties have been proven. If so, the component receives a *certificate* with respect to the certifier, which is registered in the catalog of components in an unforgeable format. Finally, the certification of a certifiable component is a pre-requisite for running it in an application.

By supporting the association of multiple certifiable components to the same certifier, SAFe makes application providers able to optimize resources, by instantiating a single parallel certification system for all of them, instead of one for each one of them. This is an important feature in a cloud computing environment.

3.2 Tactical Components

A tactical component represents a proof infrastructure, integrating a set of verification tools. It can perform a flow of execution that includes receiving a code written in the language it understands, execute validations, conversions and, finally, verify properties on such a code. Tactical components are able to exploit the parallelism features of their virtual platforms for accelerating verifications.

Besides the *LifeCycle* port, a tactical component has two other ports. Firstly, it has a user service port, with the following operations: receiving the code of the certifiable component from the certifier component, possibly previously translated by the certifier component into the language that the tactical component understands; receiving from the certifier component formal properties to be verified on that code; allowing the certifier component to monitor the progress of

the verification of properties; and returning to the certifier the result of the verification process. Secondly, it has an action port called *Verify*, containing the actions **verify_perform**, **verify_conclusive** and **verify_inconclusive**.

When **verify_perform** is activated, the tactical component starts the verification process of the formal properties assigned to it. When this process finishes, it activates **verify_conclusive**, if the verification result was conclusive for all properties (true or false), or **verify_inconclusive**, meaning that the verification of one or more properties was inconclusive (null). The verification of a property is inconclusive when the tactical component is prevented in some way from applying its verification technique to prove or refute the property. Such a situation may occur when there is some infrastructure failure on the virtual platform that places the tactical component, when the property is written in a format that is not understood by the tactical component, or when the verification timeout of the verification tool is reached. In such a case, the certifier may restart the verification process for the failed tactical component.

4 The SWC2 Certifier

If HPC Shelf can accommodate different types of component certifiers, it makes sense to consider a specific type of certifier addressing the verification of properties of the application workflow itself, rather than the functional properties of its individual components. This section introduces such a certifier — designated by SWC2 (from *Scientific Workflow Certifier Component*). Its purpose is to certify SAFeSWL workflows through the verification of a set of behavioural properties, currently resorting to a single proof infrastructure — the mCRL2 tool.

But what are the relevant properties a scientific workflow is expected to comply? The question is addressed below, based on our own experience with SAFe specifications and a recent state-of-the-art survey [9]. Other SWfMS, such as Askalon [21], BPEL Sedna [26], Kepler [20], Pegasus [12], Taverna [28] and Triana [17], were also investigated.

Firstly, scientific workflows are typically coarse-grained, due to the sort of specialized algorithms they perform. Such algorithms demand for intensive calculations, possibly taking advantage of HPC techniques and infrastructures. Coarse-grained components encapsulate most of the computational complexity of scientific workflows. Thus, orchestration languages aimed at the creation of these workflows generally offer few constructors, often limited to plain versions of sequencing, branching, iteration, parallelism and asynchronism.

On the other hand, scientific workflows are usually represented by components and execution dependencies among them, abstracting away from the computing platforms on which they run. However, during the execution of a component in a workflow, *resolution* procedures may be applied to find out which computing platform best fits its requirements. Thus, it may be interesting to verify statically if the computational actions of components are always activated after the computing platforms where they are placed have been resolved.

Scientific workflows usually adopt abstract descriptions of components, i.e. they fix only interfaces exposing available operations, without associating the component to a specific implementation. At an appropriate time of the workflow execution, a *resolution* procedure may be triggered for discovering an appropriate component implementation. Thus, it is relevant to ensure that the activation of computational actions of components is made after their effective resolution.

In order to minimize the waste of computational resources, the computing platform where a component is placed may be instantiated only when the component is strictly necessary and released when it is no longer needed. This pattern introduces three operations in the life-cycle of components: *deployment*, which installs the component implementation and possibly required libraries in the target computing platform; *instantiation*, comprising the allocation of necessary resources and configuration of the runtime environment; and, finally, *releasing*, when resources assigned to the components are deallocated. The consistency of the activation order of these operations may be statically verified. Finally, note that consistency checking of the life-cycle of components is supported by the concrete workflow certifier SWC2Impl described in Section 4.1.

But other types of workflows are also to be considered. Actually, in addition to the workflow component, which exogenously activates actions of the relevant components, each of those has an internal workflow that synchronizes with the workflow component for activating its computational operations. The *composition* of the application workflow with the internal workflows extracted from the components' code may refine the verification process, making possible to check more specialized, useful properties.

Component internal workflows recognized by HPC Shelf are those which enable/disable its actions. Each *component workflow* of a component C consists of a set of rules of the form:

$$C ::= act_1 \rightarrow act_2\downarrow \mid \top \rightarrow act_2\downarrow \mid act_1 \rightarrow act_2\nmid \qquad \text{(component rule)}$$

Let Act_C be the set of all action names of any action port of a component C, and let $act_1,\ act_2 \in Act_C$. Rule $act_1 \rightarrow act_2\downarrow$ says that when act_1 completes, act_1 is disabled and act_2 enabled. In turn, the rule $\top \rightarrow act_2\downarrow$ means that act_2 is always enabled. Finally, rule $act_1 \rightarrow act_2\nmid$ indicates that, on completion of act_1, act_1 and act_2 become disabled. Examples of component workflows are presented in the case study.

The orchestration of fine-grained components is considered too expensive in scientific workflows. In general, the time required to make the component ready (component resolution, deployment and instantiation) may exceed its effective computation time. Thus, fine-grained components with similar characteristics may be grouped into a coarse-grained component, called a *cluster* component. The activation of an action of a cluster component translates to the activation of a workflow responsible for activating each of the fine-grained components. This behavior may be incorporated into the workflow verification model for generating a more accurate specification of the computational system. Note that cluster components in HPC Shelf are parallel ones, i.e. an activation of an action of a cluster component is in fact a parallel activation of all corresponding actions in

the related fine-grained components. An example of cluster component in the case study below.

4.1 Formal Properties and Contextual Contracts

In general, properties can be divided in three classes: *default*, *application* and *ad hoc*. Default properties are common to any workflow. Typically, they include absence of deadlocks and infinite loops, and life-cycle consistency. Their verification is enabled through the following contextual signature:

$$\begin{aligned}&\text{SWC2 } [\boldsymbol{deadlock\_absence} = D : \text{DAType}, \boldsymbol{infinite\_loop\_absence} = I : \text{ILAType},\\ &\quad \boldsymbol{life\_cycle\_verification} = L : \text{LCVType}, \boldsymbol{ad\_hoc\_properties} = A : \text{AHType}]\end{aligned}$$

Context parameters ***deadlock_absence***, ***infinite_loop_absence*** and ***life_cycle_verification*** determines which sort of property is to be verified. Note that *ad hoc* properties are specified by the user resorting to the formal language supported by the certifier and stored in the workflow. The parameter ***ad_hoc_properties*** determines whether these properties are accepted.

From the contextual signature of SWC2, a contextual contract can be derived for concrete certifiers. An example is the following contract of the concrete certifier SWC2Impl:

$$\begin{aligned}&\text{SWC2Impl} : \text{SWC2 } [\boldsymbol{deadlock\_absence} = \text{DeadlockAbsence},\\ &\quad \boldsymbol{infinite\_loop\_absence} = \text{InfiniteLoopAbsence},\\ &\quad \boldsymbol{life\_cycle\_verification} = \text{LifeCycleVerification},\\ &\quad \boldsymbol{ad\_hoc\_properties} = \text{AdHocProperties}]\end{aligned}$$

This means that SWC2Impl verifies deadlock absence, infinite loop absence and life-cycle consistency, and accepts *ad hoc* properties. To accomplish this it currently orchestrates a single tactical component, mCRL2, which extends the contextual signature Tactical with a parameter ***version*** specifying a version of the mCRL2 toolset:

$$\begin{aligned}&\text{Tactical } [\boldsymbol{message\_passing\_interface} = M : \text{MPIType},\\ &\quad \boldsymbol{number\_of\_nodes} = N : \text{Integer}, \boldsymbol{number\_of\_cores} = C : \text{Integer}]\end{aligned}$$

$$\begin{aligned}&\text{mCRL2 } [\boldsymbol{version} = V : \text{VersionType}, \boldsymbol{message\_passing\_interface} = M : \text{MPIType},\\ &\quad \boldsymbol{number\_of\_nodes} = N : \text{Integer}, \boldsymbol{number\_of\_cores} = C : \text{Integer}]\\ &\quad \text{extends Tactical } [\boldsymbol{message\_passing\_interface} = M,\\ &\quad\quad \boldsymbol{number\_of\_nodes} = N, \boldsymbol{number\_of\_cores} = C]\end{aligned}$$

The parameter ***message_passing_interface*** configures the message passing library used by the tactical component (e.g. MPI [13]). In turn, ***number_of_nodes*** and ***number_of_cores*** specifies the (minimum) number of processing nodes and cores per node that will be required for execution of the tactical component. The orchestration performed by SWC2Impl is governed by the TCOL fragment depicted in Figure 2.

Finally, mCRL2Impl is declared as a concrete tactical component encapsulating version 201409.1 of the mCRL2 toolset, implemented through the MPICH2 library [3] and resorting to at least four cores per processing node:

[3] http://www.mpich.org/

```
<sequence>
    <perform action="resolve"             id_port="mCRL2-life-cycle"/>
    <perform action="deploy"              id_port="mCRL2-life-cycle"/>
    <perform action="instantiate"         id_port="mCRL2-life-cycle"/>
    <perform action="verify_perform"      id_port="mCRL2-verify"/>
    <perform action="release"             id_port="mCRL2-life-cycle"/>
</sequence>
```

Fig. 2: The orchestration code of the certifier SWC2Impl in TCOL.

$$\mathsf{mCRL2Impl} : \textsc{mCRL2}\ [\boldsymbol{version} = 201409.1,\ \boldsymbol{message\_passing\_interface} = \text{MPICH2},\ \boldsymbol{number\_of\_cores} = 4]$$

In SWC2Impl, the contextual contract of the inner mCRL2 tactical component makes mCRL2Impl a possible candidate, since it only requires MPI:

$$\textsc{mCRL2}\ [\boldsymbol{message\_passing\_interface} = \text{MPI}]$$

For certifying a workflow, the provider must create, using the architectural subset of SAFeSWL, a certification binding between the workflow component and a SWC2 component, represented by a contract like

$$\text{SWC2}\ [\boldsymbol{deadlock\_absence} = \textsc{DeadlockAbsence},\ \boldsymbol{ad\_hoc\_properties} = \textsc{AdHocProperties}]$$

This contract declares that the provider looks for a certifier that verifies deadlock absence and accepts *ad hoc* properties. Clearly, SWC2Impl is a candidate.

The certifier component determines which default and application properties are either optional or mandatory, and providers determine this for *ad hoc* ones.

5 Translating SAFeSWL to mCRL2

The verification of a SAFeSWL workflow requires its translation to the specific notation of the tactical component which will take care of it. As explained above, mCRL2 [15, 16] was chosen here to support workflow verification. System behaviors in mCRL2 are specified in a process algebra reminiscent of ACP [4]. Processes are built from a set of user-declared actions and a small number of combinators including *multi-action* synchronization, *sequential*, *alternative* and *parallel* composition, and abstraction operators (namely, action *relabeling*, *hiding* and *restriction*). Actions can be parameterized by data and conditional constructs, giving support to conditional, or data-dependent, systems' behaviors. Data is defined in terms of abstract, equational data types [22]; behaviors, on the other hand, are given operationally resorting to labeled transition systems.

mCRL2 provides a modal logic with fixed points, extending Kozen's propositional modal μ-calculus [19] with data variables and quantification over data domains. The flexibility attained by nesting least and greatest fixpoint operators with modal combinators allows for the specification of complex properties. For simplifying formulas, mCRL2 allows the use of regular expressions over the set of actions as possible labels of both necessity and eventuality modalities. The use of regular expressions provides a set of macros for property specification which are enough in practical situations.

$$
\begin{array}{lll}
T ::= L \mid G \mid T_1; T_2 \mid T_1 || T_2 \mid \texttt{repeat}\ T & & \text{(task)} \\
L ::= act \mid \texttt{break} \mid \texttt{continue} \mid \texttt{start}(h, act) \mid \texttt{wait}(h) \mid \texttt{cancel}(h) & & \text{(literal)} \\
G ::= act{\downarrow}T \ \mid \ act{\downarrow}T + G & & \text{(guarded tasks)}
\end{array}
$$

Fig. 3: Formal Grammar of the Orchestration Subset of SAFeSWL.

$$
\frac{state \xrightarrow{x} state'' \quad state'' \overset{xs}{\Longrightarrow} state'}{state \xrightarrow{x \cdot xs} state'} \ \text{(big-step)}
\qquad
\frac{a \in E}{\langle a, T, E, L, S, F\rangle \xrightarrow{a} \langle T, \texttt{stop}, E, L, S, F\rangle} \ \text{(action)}
$$

$$
\frac{\langle T_1, \texttt{stop}, E, L, S, F\rangle \overset{xs}{\Rightarrow} \langle T_1', R_1, E', L', S', F'\rangle}{\langle T_1 || T_2, T, E, L, S, F\rangle \overset{xs}{\Rightarrow} \langle (T_1'; R_1 || T_2, T, E', L', S', F'\rangle} \ \text{(parallel-left)}
\qquad
\frac{}{\langle \texttt{stop} || T_2, T, E, L, S, F\rangle \rightarrow \langle T_2, T, E, L, S, F\rangle} \ \text{(stop-par-left)}
$$

$$
\frac{a \in E}{\langle a{\downarrow}T_1 + G, T_2, E, L, S, F\rangle \rightarrow \langle T_1, T_2, E, L, S, F\rangle} \ \text{(select-left)}
\qquad
\frac{a \notin E \quad \langle G, T_2, E, L, S, F\rangle \rightarrow \langle T_3, T_2, E, L, S, F\rangle}{\langle a{\downarrow}T_1 + G, T_2, E, L, S, F\rangle \rightarrow \langle T_3, T_2, E, L, S, F\rangle} \ \text{(select-right)}
$$

$$
\frac{}{\langle (T_1; T_2), T, E, L, S, F\rangle \rightarrow \langle T_1, (T_2; T), E, L, S, F\rangle} \ \text{(sequence)}
\qquad
\frac{}{\langle \texttt{repeat}\ T_1, T_2, E, L, S, F\rangle \rightarrow \langle T_1, \texttt{stop}, E, (T_1, T_2) \cdot L, S, F\rangle} \ \text{(repeat)}
$$

$$
\frac{}{\langle \texttt{continue}, T, E, (T_i, T_f) \cdot L, S, F\rangle \rightarrow \langle T_i, T, E, (T_i, T_f) \cdot L, S, F\rangle} \ \text{(continue)} \quad [2\text{mm}]
\qquad
\frac{}{\langle \texttt{break}, T, E, (T_i, T_f) \cdot L, S, F\rangle \rightarrow \langle T_f, \texttt{stop}, E, L, S, F\rangle} \ \text{(break)}
$$

$$
\frac{a \in E \quad fresh(h)}{\langle \texttt{start}(a, h), T, E, L, S, F\rangle \xrightarrow{start(a,h)} \langle T, \texttt{stop}, E, L, S \cup \{(a, h)\}, F\rangle} \ \text{(start)}
\qquad
\frac{h \in F}{\langle \texttt{wait}(h), T, E, L, S, F\rangle \rightarrow \langle T, \texttt{stop}, E, L, S, F\rangle} \ \text{(wait)}
$$

$$
\frac{(a, h) \in S}{\langle T_1, T_2, E, L, S, F\rangle \xrightarrow{(a,h)} \langle T_1, T_2, E, L, S - \{(a, h)\}, F \cup \{h\}\rangle} \ \text{(finish)}
\qquad
\frac{(a, h) \in S}{\langle \texttt{cancel}(h), T, E, L, S, F\rangle \rightarrow \langle T, \texttt{stop}, E, L, S - \{(a, h)\}, F\rangle} \ \text{(cancel)}
$$

Note: rules (parallel-right) and (stop-par-right) are omitted in this figure, since they are symmetric to (parallel-left) and (stop-par-left), respectively.

Fig. 4: Operational semantics of the orchestration subset of SAFeSWL.

5.1 The Translation Process

The translation process follows directly the operational rules (Figure 4) defined for a formal version of the orchestration subset of SAFeSWL (Figure 3).

Let W be the workflow component of a parallel computing system. In such a grammar, c ranges over component identifiers, h ranges over naturals and $act \in Act_W$. For each component, we assume a minimal set of workflow actions, including life cycle ones ($\{\texttt{resolve}_c, \texttt{deploy}_c, \texttt{instantiate}_c, \texttt{release}_c\} \subseteq Act_W$).

The semantics of W consists of a task T_W, given by the rules in Figure 4, and initial state $\langle T_W, \texttt{stop}, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. The symbol `stop` denotes task completion. Each execution state consists of a tuple $\langle T_1, T_2, E, L, S, F \rangle$, where T_1 is the next task to be evolved; T_2 is the following task to be evolved; E are the actions enabled in the components; L is a stack of pairs with the beginning and the end of the repeat blocks scoping the current task; S is a set of pairs with actions asynchronously activated that have not yet been finished and their handles; and F is a set of handles associated to finished asynchronous actions.

For simplicity, the behavior imposed by the internal workflows of components is omitted, which enable/disable their actions and directly manipulate E.

Rule `big-step` denotes a big-step transition relation between execution states. Rule `action` states that a state containing the activation of an enabled action causes the system to observe the action and go to the state in which the next task is evaluated. Rule `sequence` indicates sequential evaluation of tasks. Rule (`parallel-left`) states that if a state X with a task T_1 leads to any state Y in any number of steps, the parallelization of T_1 with a task T_2, starting from X, leads to Y, however propagating the parallelism to the next task. Rule `stop-par-left` denotes parallel termination (join). Rule `select-left` indicates that the activated action must be enabled. Rule `select-right` states that a disabled action may not be activated. Rule `repeat` performs a task T_1 and stores in L the iteration beginning and end tasks, which are performed, respectively, through rules `continue` and `break`. Rule `start` says that an enabled action and a handle not yet used can be associated and added to S, emitting an action to the system ($start(A, h)$). Rule `finish` indicates that an action asynchronously activated can actually occur, having its handle registered in F and emitting an action to the system ((*a*,*h*)). Rule `wait` states that waiting for a finished asynchronous action has no effect. Finally, rule `cancel` cancels an asynchronous action.

We may now briefly present an informal description of the translation process. Rule (`action`) states that every SAFeSWL action is an observable mCRL2 action. Rule (`sequence`) states that a sequence of two tasks in SAFeSWL is translated by the sequential composition of the corresponding translations. Rules (`parallel-left`) and (`parallel-right`) mean that the translation of a set of parallel tasks takes place by the creation of mCRL2 processes in a fork-join paradigm. Rules (`select-left`) and (`select-right`) indicate the need for the creation of mCRL2 processes that control the state (enabled/disabled) of actions. Rules (`repeat`), (`continue`) and (`break`) indicate, respectively, the need for the creation of a mCRL2 process that manages a repetition task in order to detect the need for a new iteration, the return back to the beginning of the

iteration, or the end of the iteration. Rule (`start`) states the need for the creation of an asynchronous mCRL2 process that will eventually perform the action. Moreover, it is also needed to create a manager process that stores the state of all actions started asynchronously (pending or finished). Finally, rules (`wait`) and (`cancel`) indicate the need for the communication with such a manager to, depending on the state of the asynchronous action, block the calling process or cancel the asynchronous process launched for the action.

5.2 Default Properties in mCRL2

The first default property is deadlock absence, specified as DA : $[\mathsf{true}*]\langle\mathsf{true}\rangle\mathsf{true}$, i. e. there is always a possible next action at every point in the workflow.

A workflow that contains a **repeat** task may perform an infinite loop when a **break** is not reachable within its scope. We may check infinite loop absence (ILA) by verifying if all mCRL2 $break(i)$ actions can occur from a certain point on, where i is the index of the related **repeat** task, using the following formula:

$$\mathsf{ILA}: \ \forall i : Nat.[\mathsf{true}*]\langle\mathsf{true} * .break(i)\rangle\mathsf{true}$$

The remaining properties express life-cycle restrictions in terms of precedence relations specified by formulas like

$$\begin{aligned}\mathsf{LC1}: \ &\forall c : Nat.[!resolve(c) * .deploy(c)]\mathsf{false}\\ &\&\& \ \langle\mathsf{true} * .resolve(c).!release(c) * .deploy(c)\rangle\mathsf{true}\end{aligned}$$

This formula is applied to each component c, restricted to orchestrated components in order to reduce the model checking search space. The first part of the conjunction states that a **deploy** may not be performed before a **resolve**. Note that ! stands for set complement, thus the expression $[!a]\,\mathsf{false}$ states that all evolutions by an action different from a are forbidden. The second part states that a **deploy** may be performed, since a **resolve** has been performed before and there is not a **release** between **resolve** and **deploy**. Similar restrictions may be specified for different pairs of life-cycle actions using a similar pattern, such as:

$$\begin{aligned}\mathsf{LC2}: \ &\forall c : Nat.[!deploy(c) * .instantiate(c)]\mathsf{false}\\ &\&\& \ \langle\mathsf{true} * .deploy(c).!release(c) * .instantiate(c)\rangle\mathsf{true}\\ \mathsf{LC3}: \ &\forall c, a : Nat.[!instantiate(c) * .compute(c,a)]\mathsf{false}\\ &\&\& \ \langle\mathsf{true} * .instantiate(c).!release(c) * .compute(c,a)\rangle\mathsf{true}\\ \mathsf{LC4}: \ &\forall c : Nat.[!instantiate(c) * .release(c)]\mathsf{false}\\ &\&\& \ \langle\mathsf{true} * .instantiate(c).!release(c) * .release(c)\rangle\mathsf{true}\end{aligned}$$

Here, **compute**(c, a) represents the computational action a of a component c, declared in the architectural description of the workflow.

6 A Case Study

MapReduce is a programming model implemented by a number of large-scale data parallel processing frameworks, firstly proposed by *Google Inc.* [11]. A user must specify: a *map function*, which is applied by a set of parallel *mapper* processes

to each element of an input list of *key/value* pairs (KV-pairs), returning a set of elements in an intermediary list of KV-pairs; and a *reduce function*, which is applied by a set of parallel *reducer* processes to all intermediate values associated with the same *key* across all mappers, yielding a list of output pairs.

Figure 5 depicts a simple example of MapReduce for processing a text containing lines of words *green*, *yellow*, *blue* and *pink*. At the end of the processing, the expected output is the number of occurrences of each color in the text.

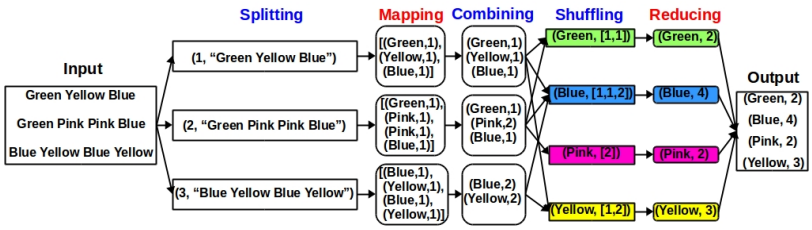

Fig. 5: A classic example of word counting with MapReduce.

We have designed a framework of components for MapReduce computations in HPC Shelf, comprising the following components: DataSource, a *data source* that stores the input data structure; Mapper, a *computation* that implements a set of parallel mapping agents; Reducer, a *computation* that implements a set of parallel reducing agents; DataSink, a *data source* that stores the output data structure, generated from the output pairs produced by the reducing agents; Splitter, a *connector* that takes the list of input pairs from the data source (first iteration) and outputs pairs generated by the reducing agents (produced in the previous iteration) and either distributes them to the mapping agents (to start a new iteration) or sends them to the data sink (to end the process); and Shuffler is a *connector* that groups intermediate keys produced by the mapping agents and redistributes them among the reducing agents.

Figure 6 depicts the architecture of a simple iterative MapReduce parallel computing system. Each computation/connector has three ports: a user and provider service port, from which they input KV-pairs and output KV-pairs, respectively; and a single *action* port, called *TaskChunk*, for orchestrating tasks. KV-pairs are transmitted (between splitter and shuffler) and processed (by mapper and reducer) incrementally for optimizing communication granularity and overlapping mapping and reducing phases in the same iteration.

TaskChunk has the three action names: **chunk_ready**, for signaling that a new chunk of KV-pairs is available; **read_chunk**, for inputing KV-pairs from the next chunk; **perform**, for processing the KV-pairs read from a chunk.

For our purposes, the contextual signatures of the abstract components and the contextual contracts of the concrete components, whose instances were presented, can be omitted. The SAFeSWL workflow script will be not shown here, but it is available from `http://www.lia.ufc.br/~allberson/swc2`.

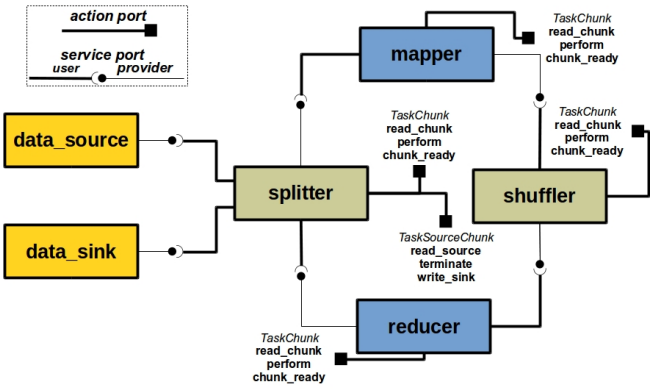

Fig. 6: The MapReduce architecture.

6.1 Internal Workflows of MapReduce Components

The following simplified version of the internal workflows of MapReduce components is enough for proving all properties relevant to this case study:

$$
\begin{aligned}
shuffler = mapper = reducer = combiner = \\
\quad \{\top \to resolve\downarrow, \top \to deploy\downarrow, \top \to instantiate\downarrow, \\
\quad \top \to read\_chunk\downarrow, \top \to perform\downarrow,\ perform \to chunk\_ready\downarrow\} \\
splitter = shuffler\ \cup\ \{\top \to read\_source\downarrow, \top \to write\_sink\downarrow, \\
\quad perform \to terminate\downarrow\}
\end{aligned}
$$

6.2 Certification of the MapReduce Workflow

Consider the certification architecture depicted in Figure 7. It contains a certification binding linking the MapReduce workflow to SWC2, containing the valuation described in Section 4.1, which aims to choose SWC2Impl. When the application is started by SAFe, a parallel certification system is automatically instantiated and executed. During the certification process, SWC2Impl was chosen, with mCRL2Impl, also described in Section 4.1, as its tactical component, over a virtual platform containing 4 processing nodes. The result of the translation of the MapReduce workflow into mCRL2 is available at `www.lia.ufc.br/~allberson/swc2`.

6.3 MapReduce *Ad Hoc* Properties

The MapReduce *ad hoc* properties include a *safety* and a *liveness* group. The former describes precedences of execution between two distinct components or component actions. Two examples, among a list of 11, are shown for illustrative

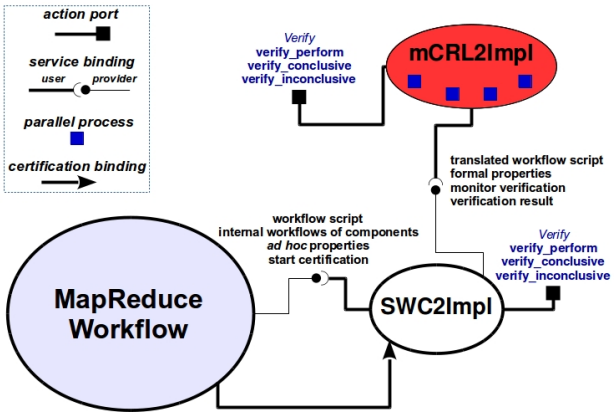

Fig. 7: Certification architecture for the MapReduce workflow.

purposes. Note that the numbers in the formulas correspond to specific component and action identifiers, as they appear in the original SAFeSWL code.

$$\begin{aligned} \text{SbM}: \ & [!compute(3,352) * .compute(5,551)]\text{false} \\ \text{MbC}: \ & [\text{true} * .compute(5,551).!compute(3,352) * . \\ & compute(5,551)]\text{false} \end{aligned}$$

Property SbM states that the action **read_chunk** (551), of mapper (5), must be preceded by the action **perform** (352), of splitter (3). On its turn, the second property expresses that the action **perform** (352), of splitter (3), must occur between two executions of the action **read_chunk** (551), of mapper (5).

The liveness group includes a broader ontology of properties. For example,

$$\begin{aligned} \text{LIV1}: \ & \langle true * .guard(3,342)\rangle \text{true} \\ \text{LIV2}: \ & \forall c : Nat, a : Nat.\, \nu X.\mu Y.[compute(c,a)]Y \,\&\&\, [!compute(c,a)]X \\ \text{LIV3}: \ & [\text{true}*](\forall c : Nat, a : Nat => \mu Y.([!compute(c,a)]Y \,\&\&\, < \text{true} > \text{true})) \end{aligned}$$

Property LIV1 ensures the existence of workflow traces including the execution of a particular action (**terminate** (342), of splitter (3)). The second property expresses the fact that an action can only be executed along non consecutive periods. Finally, LIV3 is a non starvation condition, that is, for every reachable state it is possible to execute **compute**(c, a), for any possible value of c and a. Note the quantification over the workflow components and actions.

The 20 formal properties (both default and *ad hoc*) verified were distributed among the 4 units of the tactical component, and proven. Figure 8 depicts the average times for the certification of the MapReduce workflow, by varying the number of processing nodes and processing cores per node. As expected a judicious use of parallelism can cut the execution time in half in most cases.

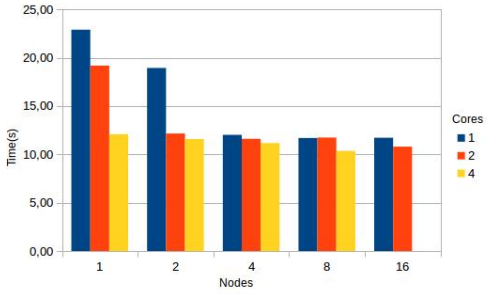

Fig. 8: MapReduce workflow certification times.

7 Conclusions and Future Work

In the previous sections, we have made a case for the introduction, within a platform for orchestrating parallel components, of a specific component whose purpose is to verify behavioral properties of workflow that link different components which make up an application. This has a *reflexive* character: a component that contributes to certify the emergent orchestrated behavior in an application.

This idea was made concrete in the context of the HPC Shelf platform, in which we characterized SWC2 – a scientific workflow certifier. The role of SWC2 within an application is clear: it increases the confidence on the underlying workflow specification and may avoid anomalous or erroneous behaviours introduced by design errors. Moreover, a relevant characteristic of the architecture proposed for the certification process resides in the fact that new tactical components and workflow certifiers can be smoothly added, to deal with the verification of different classes of properties. For example, the development of new tactical components for SWC2, encapsulating other verification tools, such as Uppaal or Interactive Markov Chains, to deal with time constraints or probabilistic behavior, respectively, is part of our current work.

The workflow certifier in HPC Shelf, as well as the underlying certification framework, is still an ongoing project. However, an initial prototype was developed in C#/MPI and validated through a number of benchmark examples in the development of scientific computing applications. The example discussed in this paper — MapReduce, available from `github.com/UFC-MDCC-HPC/HPC-Shelf-Certification` — provides an interesting illustration.

In any case, the relevant message is conceptual, going beyond its concrete implementation in the HPC Shelf platform. Actually, we believe that the notions of workflow certifier, tactical component and parallel certification system can be successfully employed in the certification of workflows in widespread SWfMS, such as Pegasus or Taverna.

References

1. Wil M. P. Van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426. Springer-Verlag, 1997.
2. Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.
3. Paul Attie, Munindar Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings.*, pages 134–145, 1993.
4. J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational theories of communicating processes.* Cambridge Tracts in Theoretical Computer Science (50). Cambridge University Press, 2010.
5. Hasan Davulcu, Michael Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 25–33. ACM, 1998.
6. Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD Record*, volume 19, pages 204–214. ACM, 1990.
7. F. H. de Carvalho Junior, R.D Lins, R. C. Correa, and G. A. Araújo. Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience*, 19(5):697–719, 2007.
8. Francisco Heron de Carvalho Junior, C. A. Rezende, J. C. Silva, and W. G. Al Alam. Contextual abstraction in a type system for component-based high performance computing platforms. *Science of Computer Programming*, 2016.
9. J. de Carvalho Silva and F. H. C. de Carvalho Junior. A platform of scientific workflows for orchestration of parallel components in a cloud of high performance computing applications. In *Programming Languages - 20th Brazilian Symposium, SBLP 2016, Maringá, Brazil, September 22-23, 2016, Proceedings*, pages 156–170, 2016.
10. A. B. de Oliveira Dantas, F. H. de Carvalho Junior, and L. Soares Barbosa. A framework for certification of large-scale component-based parallel computing systems in a cloud computing platform for hpc services. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pages 229–240. ScitePress, 2017.
11. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
12. Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
13. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. An Introduction to the MPI Standard. Technical Report CS-95-274, University of Tennessee, January 1995.
14. Xiang Fu, Tevfik Bultan, Richard Hull, and Jianwen Su. Verification of vortex workflows. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, pages 143–157, 2001.

15. J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mcrl2. In *Methods for Modelling Software Systems: Dagstuhl Seminar 06351*, 2007.
16. J. F. Groote and M. R.. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
17. Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. Ws-rf workflow in triana. *International Journal of High Performance Computing Applications*, 22(3):268–283, 2008.
18. Richard Hull, Francois Llirbat, Eric Siman, Jianwen Su, Guozhu Dong, Bharat Kumar, and Gang Zhou. Declarative workflows that support easy modification and dynamic browsing. *ACM SIGSOFT Software Engineering Notes*, 24(2):69–78, 1999.
19. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, (27):333–354, 1983.
20. Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.
21. Jun Qin, Thomas Fahringer, and Sabri Pllana. Uml based grid workflow modeling under askalon. In *Proceedings of the Distributed and Parallel Systems: From Cluster to Grid Computing (DAPSYS 2006)*. Springer, September 2006.
22. D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge University Press, 2011.
23. Pinar Senkul, Michael Kifer, and Ismail H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 694–705. VLDB Endowment, 2002.
24. Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
25. Wil M. P. Van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.
26. Bruno Wassermann, Wolfgang Emmerich, Ben Butchart, Nick Cameron, Liang Chen, and Jignesh Patel. *Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling*, pages 428–449. Springer London, London, 2007.
27. Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *International Conference on Database Theory*, pages 230–246. Springer, 1997.
28. Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557, 2013.