Science of Computer Programming 191 (2020) 102379

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A component-based framework for certification of components in a cloud of HPC services

Allberson Bruno de Oliveira Dantas [a], Francisco Heron de Carvalho Junior [b,*], Luis Soares Barbosa [c]

[a] *IEAD, Universidade da Integração Internacional da Lusofonia Afro-Brasileira, Campus da Liberdade, Redenção, Brazil*
[b] *Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Brazil*
[c] *HASLab INESC TEC & Universidade do Minho, Campus de Gualtar, Braga, Portugal*

ARTICLE INFO

Article history:
Received 16 March 2018
Received in revised form 29 November 2019
Accepted 20 December 2019
Available online 2 January 2020

Keywords:
Verification-as-a-Service
Formal verification
High Performance Computing
Software components
Scientific workflows

ABSTRACT

HPC Shelf is a proposal of a cloud computing platform to provide component-oriented services for High Performance Computing (HPC) applications. This paper presents a Verification-as-a-Service (VaaS) framework for component certification on HPC Shelf. Certification is aimed at providing higher confidence that components of parallel computing systems of HPC Shelf behave as expected according to one or more requirements expressed in their contracts. To this end, new abstractions are introduced, starting with certifier components. They are designed to inspect other components and verify them for different types of functional, non-functional and behavioral requirements. The certification framework is naturally based on parallel computing techniques to speed up verification tasks.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

HPC Shelf is a cloud computing platform aimed at addressing domain-specific, computationally intensive problems typically emerging from computational science and engineering domains. For this purpose, it provides a range of High Performance Computing (HPC) services based on *parallel computing systems*. They are built from the orchestration of parallel components representing both software and hardware elements of HPC systems. The hardware elements represent distributed memory parallel computing platforms such as clusters and MPPs.[1] Software components, representing parallel computations, attempt to extract the best performance of them.

Parallel computing systems are managed by SAFe (Shelf Application Framework) [1]. By means of SAFe, *application providers* build *applications*, through which *domain specialists* access the services of HPC Shelf. Applications are domain-specific problem-solving environments, such as *web portals* [2]. They provide a high-level interface through which specialists specify problems. Computational solutions to these problems are automatically generated according to rules programmed by application providers, in the form of parallel computing systems.

Application providers must have technical background to create computational solutions to problems in their application domains. In HPC Shelf, they must be able to identify and combine components to form parallel computing systems. Thus, background on parallel computing platforms and programming for such platforms is not required for application providers.

* Corresponding author.
E-mail addresses: allberson@unilab.edu.br (A.B. de Oliveira Dantas), heron@lia.ufc.br (F.H. de Carvalho Junior), lsb@di.uminho.pt (L.S. Barbosa).

[1] Massively Parallel Processing systems.

https://doi.org/10.1016/j.scico.2019.102379
0167-6423/© 2020 Elsevier B.V. All rights reserved.

This is a requirement for *component developers*. Combined with the inherent complexity of parallel system design, this fact implies the need for effective mechanisms to ensure that, in parallel computing systems, components and interactions between them behave as expected by application providers and predicted by component developers. In software engineering, this is a problem known as *certification of software components* [3–7]. In the context of HPC Shelf, the certification problem can be seen from two perspectives. In the component perspective, each component implementation is verified against the functional, non-functional, and behavioral requirements declared in its published interface. In the system perspective, typical safety and liveness properties of the component orchestration workflow should be ensured.

The need for rigorous validation, leading to some form of system certification, brings formal methods into the picture in order to identify, and possibly rule out, faulty behaviors in applications. Such methods are, however, not so common in the domain of HPC systems, due to their inherent complexity and the difficulty of their concrete implementation. Indeed, HPC systems are defined by heterogeneous computing platforms composed concurrently. In fact, in spite of more than three decades of active research in formal methods for software development and verification, we are still far from what should be the practice of a true engineering discipline, supported by a stable and sound mathematical basis. In most cases, testing and *a posteriori* empirical error detection are still dominant, even in scenarios where formal verification is a requirement (e.g. safety-critical systems).

The work reported in this article presents the proposal for a cloud-based general-purpose certification framework for HPC Shelf. Through the proposed framework, components called *certifiers* may use a set of different certification tools to certify that the components of parallel computing systems meet a certain set of requirements. The case studies used to demonstrate the proposed certification framework are particularly focused on functional and behavioral requirements that can be verified through automated verification methods and tools, such as theorem provers and model checkers. The certification process becomes integrated with the parallel computing systems in a highly modular way, so that new certifier components may be inserted according to the verification tasks required in the certification process.

The certification process may be carried on in parallel. For this, certifier components are defined as *parallel certification systems*, analogous to parallel computing systems. Parallel certification systems contain a *certification-workflow component* and a set of *tactical components*, each one providing the access to an existing certification tool or infrastructure running in a parallel computing platform. Thus, within tactical components, parallel computing may help exploit the maximum performance of the underlying verification infrastructures to accelerate the certification process.

Summing up, the main artifacts produced by the work whose results are reported in this article are the following ones:

- A *general-purpose certification framework* for HPC Shelf;
- A class of *certifier components*, named C4, for the certification of computation components of HPC Shelf;
- Another class of *certifier components*, named SWC2, for the certification of workflow components;
- A set of *tactical components* to make the bridge between the above certifier components and existing formal verification tools.

C4 and SWC2 have the purpose of helping proof-of-concept validation of the certification framework of HPC Shelf. It has also been evaluated in the context of other cloud-based software certification initiatives, with emphasis on works related to VaaS, HPC, and automatic software-verification tools. From this assessment, the following outstanding features and contributions have been identified in favor of the certification framework of HPC Shelf:

- It is general purpose, in the sense that it is not intended to certify a particular requirement, although the case studies presented in this article focus on the verification of functional and behavioral properties through deductive program verification and model checking tools.
- It does not certify only software components, but any kind of component, including components representing hardware elements, such as parallel computing platforms in HPC Shelf.
- It is fully component-oriented, with seamless integration with the environment, in the sense that certification is introduced by certifier and tactical components that may encapsulate certification tools.
- It is the first certification framework in the context of component-based high performance computing (CBHPC), where certification may avoid the wasting of time and financial resources due to delays, crashes, and wrong outputs in the execution of long-running computations.
- It is the first VaaS framework applied in the context of HPC.
- It introduces new ideas for VaaS framework design, such as:
  - the use of component-orientation to support a higher level of abstraction with respect to underlying formal verification tools;
  - the clear role separation among certification authorities, component developers, and system builders (application providers);
- It presents a general method for exploring parallel processing to speedup certification tasks at several levels, by exploring the parallel computing infrastructure where parallel components subject to certification runs.

Article structure. After a description of HPC Shelf in Section 2, its certification framework is introduced in Section 3. Sections 4 and 5 detail, respectively, the architecture of C4 and SWC2 certifiers. Next, Section 6 presents some case studies to

demonstrate the use of C4 and SWC2 certifiers in parallel computing systems. A discussion about related works obtained through a systematic search in scientific databases is presented in Section 7, with emphasis in certification of component-based software systems and VaaS. Finally, Section 8 presents concluding remarks, pointing to further works.

2. HPC Shelf

HPC Shelf is a cloud computing platform that provides HPC services for *providers* of domain-specific *applications*. An application is a problem-solving environment through which *specialist users*, the end users of HPC Shelf, specify problems and obtain computational solutions for them. It is assumed that these solutions are computationally intensive, thus demanding the use of large-scale parallel computing infrastructure, i.e. comprising multiple parallel computing platforms engaged in a single computational task.

Applications generate computational solutions as component-oriented *parallel computing systems*. They are built by composition and orchestration of a set of parallel components that represent hardware and software elements, addressing functional and non-functional concerns. To do so, these components comply to Hash [8], a parallel component model whose components may exploit parallel processing in distributed-memory parallel computing platforms. For that, they are formed by a set of units, each one placed on one of their processing nodes, so that parallelism concerns may be confined to a single component. Parallel components that are compatible with Hash may be combined hierarchically through *overlapping composition* [9].

2.1. Component kinds of parallel computing systems

Component platforms that comply to the Hash component model distinguish components according to a set of *component kinds*. A component kind represents a set of components with similar deployment and interaction models, possibly representing building-block abstractions of an application domain. HPC Shelf supports the following component kinds:

- ***virtual platforms***, representing distributed-memory parallel computing platforms (e.g. clusters and MPPs);
- ***computations***, representing parallel algorithm implementations that may accelerate performance by exploiting the architectural characteristics and features of a class of virtual platforms;
- ***data sources***, representing repositories of data required by computations, possibly BigData ones;
- ***connectors***, formed by a set of *facets*, co-located with computations and data sources located at different virtual platforms, aimed at orchestrating them and/or supporting choreographs involving them;
- ***service bindings***, aimed at binding *user* and *provider* ports exported by components for communication with the environment and among them, as well as supporting non-functional services;
- ***action bindings***, aimed at binding *action* ports of computations and connectors for orchestration of computational, communication and synchronization tasks supported by them.

Through a *service binding*, a component may consume a service offered by another component. This is only possible if the former component has a *user port* and the later one has a *provider* port, whose types must be compatible with the type of the service binding. Therefore, the service binding may act either as a simple service dispatcher or as a service adapter.

Action bindings connect a set of action ports belonging to computation and connector components. These ports are typed by a set of *action names*, which label computational, communication or synchronization tasks. A component may export one or more *action ports*. The orchestration of action names defines the *workflow* of a parallel computing system.

An action port offers, to computation and connectors, an interface to control the *activation* of action names (Listing 1). The activation of an action name n completes in an action binding if there is a pending activation of n in each action port. Otherwise, it remains blocked.

The components of HPC Shelf have a default lifecycle action port, referred with the own component instance identifier in a parallel computing system. It has the following reflexive action names:

- **resolve**, which attempts to select a component implementation that best fits the requirements of its *contextual contract* (Section 2.5);
- **deploy**, which deploys the selected component implementation in the virtual platform where it will be instantiated for starting execution;
- **instantiate**, which makes a component ready for computation and interaction with other components through its service and action ports;
- **run**, which starts the internal orchestration logic of the component, i.e. the orchestration of action names of their action ports;
- **release**, which releases the resources allocated for the component instance, after its computation has finished.

In the workflow of a parallel computing system, the activation of the lifecycle action names must respect the following protocol (regular expression):

```
public interface ITaskBinding : IActivateKind , GoPort
{
    //synchronous , simple , no reaction code
    void invoke(object action );
    //synchronous , multiple , no reaction code
    object invoke(object[] action );
    //asynchronous , simple , no reaction code
    void invoke(object action , out IActionFuture f );
    //asynchronous , multiple , no reaction code
    void invoke(object[] action , out IActionFuture f );
    //asynchronous , simple , reaction code
    void invoke(object action , Action reaction , out IActionFuture f );
    //asynchronous , multiple , reaction code
    void invoke(object[] action , Action[] reaction , out IActionFuture f );
}

public interface IActionFuture
{
    void wait();                    // waits for completion of the action
    bool test();                    // tests the completion of an action
    IActionFutureSet createSet();   // creates a set of pending actions
    object Action { get; }          // retrieves the reaction code to be executed
    void registerWaitingSet (AutoResetEvent waiting_set );
    void unregisterWaitingSet (AutoResetEvent waiting_set );
}

public interface IActionFutureSet: IEnumerable<IActionFuture>
{
    void addAction(IActionFuture f ); // add a new activation action
    void waitAll();                   // waits for completion of all pending actions
    IActionFuture waitAny();          // waits for completion of any pending action
    bool testAll();                   // tests the completion of all pending actions
    IActionFuture testAny();          // tests the completion of any pending action
    IActionFuture[] Pending { get; }  // get the list of pending actions
}
```

Listing 1: `ITaskBinding` interface (C#).

$$\textbf{resolve} \cdot \textbf{deploy} \cdot \textbf{instantiate} \cdot \textbf{run} \cdot \textbf{release} \cdot \Big((\textbf{resolve} \cdot \textbf{deploy})^? \cdot \textbf{instantiate} \cdot \textbf{run} \cdot \textbf{release} \Big)^\star$$

The protocol specifies that the workflow must resolve the contextual contract of each component before its instantiation. However, it may invoke contract resolution multiple times. After each resolution, the component must be deployed again, before instantiation. It is worth remembering that a distinct component implementation may be selected in each invocation to the contract resolution service. Finally, all instantiated components must be released before workflow termination, freeing resources.

Parallel computing systems. A parallel computing system consists of two special components, respectively called *application* and *workflow*, and a set of *solution components* of the previously described kinds. The *workflow* component represents an orchestration engine that will drive the overall computation. It may be programmed by using a general-purpose programming language (currently, C#) or SAFeSWL (SAFe Scientific Workflow Language), an XML-based orchestration language designed for activating the computational tasks of the solution components in a prescribed order [1]. Indeed, an application may generate SAFeSWL code by dynamically building computational solutions to problems specified by specialist users. Through a provenance mechanism recently developed, automatically generated parallel computing systems may be saved in the component catalog by treating them as components. The XSD grammar of SAFeSWL is presented in the Appendix A. The *application* component makes communication between the application frontend and solution components possible through service bindings.

2.2. Parallel computing systems through an example: MapReduce

MapReduce is the parallel processing model of a number of large-scale parallel processing frameworks [10]. 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-pair[2]) and returns 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 each element of an intermediate list of *key/multi-value* pairs (KMV-pair[3]), and yield a list of output KV-pairs.

Fig. 1 illustrates a MapReduce computation for counting the frequencies of words *green*, *yellow*, *blue* and *pink* in a text. At the end of the computation, the expected output is the number of occurrences of each color in the text.

A framework of components for MapReduce computations has been designed for HPC Shelf [11]. It comprises a pair of component types of kind ***computation***, named MAPPER and REDUCER, which represent mapping and reducing agents, and another pair of component types of kind ***connector***, named SPLITTER and SHUFFLER, which intermediate communication of KV/KMV-pairs among mappers, reducers and data sources.

[2] A KV-pair is a pair $\langle k, v\rangle$, where k is a key and v is a value.

[3] A KMV-pair is a pair $\langle k, [v_1\ v_2\ \dots\ v_n]\rangle$, where k is a key, mapped to values v_i, for $i \in \{1, 2, \dots, n\}$.

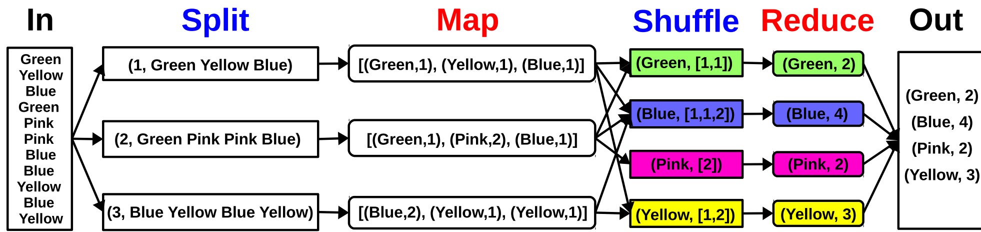

Fig. 1. A classic example of counting word frequencies with MapReduce. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

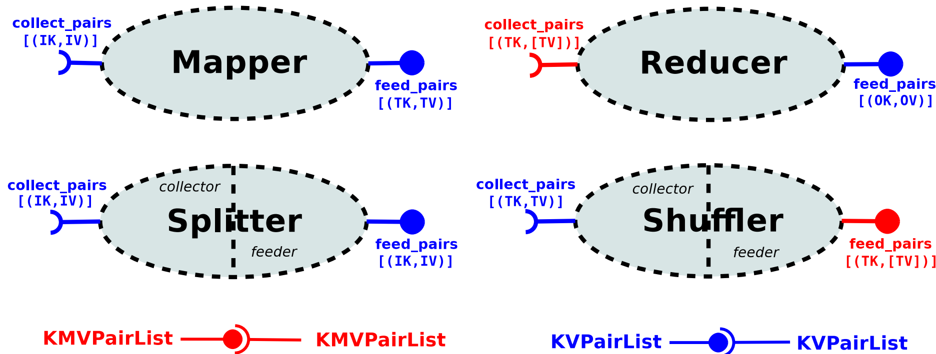

Fig. 2. The building blocks of MapReduce parallel computing systems.

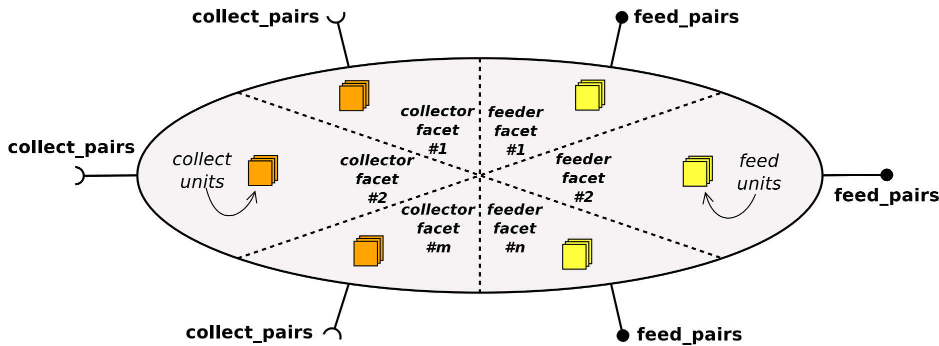

Fig. 3. Multiple facets of Splitter and Shuffler.

Fig. 2 depicts the service ports of Mapper, Reducer, Splitter and Shuffler. Their action ports will be introduced later. They have a pair of service ports named **collect_pairs** and **feed_pairs**. In the connectors, these ports are placed at different facets. Since they are *multiple facets*, instead of *single facets*, connectors may have a set of **collect_pairs** and another set of **feed_pairs** ports, as illustrated in Fig. 3. Thus, mappers, reducers, splitters and shufflers receive a list of either KV-pairs (type $[(K, V)]$, highlighted in blue) or KMV-pairs (type $[(K, [V])]$, highlighted in red) through each **collect_pairs** binding, and send a list of either KV-pairs or KMV-pairs through each **feed_pairs** binding. In particular:

- *mappers* receive a list of input KV-pairs of type (IK, IV), apply a map function to each input pair in order to yield a list of output KV-pairs of type (TK, TV), and return the list of yielded output KV-pairs;
- *reducers* receive a list of input KMV-pairs of type $(TK, [TV])$, apply a reduce function to each input pair in order to yield an output KV-pair of type $[(OK, OV)]$, and return the list of yielded output KV-pairs;
- *splitters* receive a set of lists of input KV-pairs of type (IK, IV), each one associated to a *collector* facet, redistribute the input KV-pairs across a set of output KV-pairs lists, each one associated to a *feeder* facet, and send them through the corresponding **feed_pairs** port;
- *shufflers* receive a set of lists of input KV-pairs of type (TK, TV), each one associated to a *collector* facet, group the pairs with the same key in a single KMV-pair of type $[(TK, [TV])]$, distribute the KMV-pairs across a set of output KMV-pair lists, each one associated to a *feeder* facet, and send them through the corresponding **feed_pairs** port.

In a MapReduce parallel computing system, mappers, reducers, splitters, and shufflers may be connected through bindings between their compatible **collect_pairs** and **feed_pairs** ports. For instance, Fig. 4 depicts the architecture of a simple

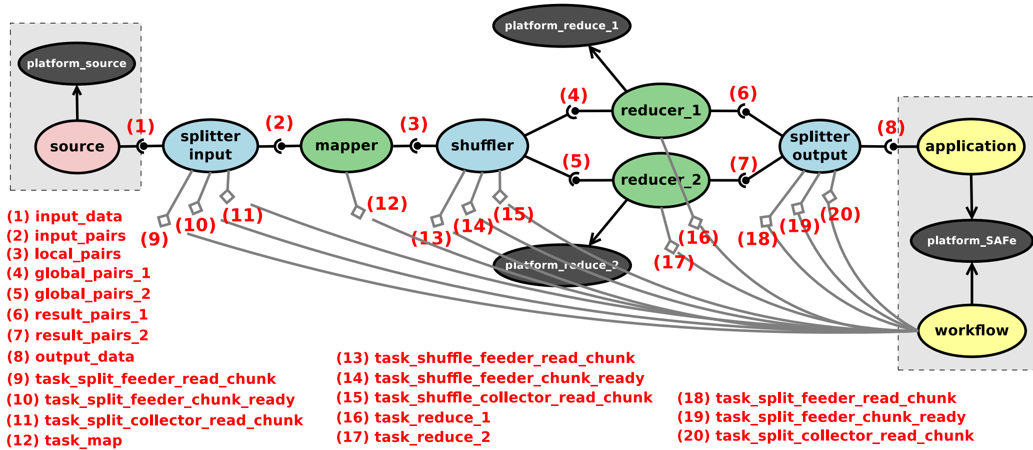

Fig. 4. The architecture of a MapReduce parallel computing system.

iterative MapReduce parallel computing system, comprising single map and reduce stages, where the reduce stage is performed by two parallel reducing agents. The input is read from a data source component (**source**) and transformed into a set of input pairs by the **input_data** binding. In turn, the output pairs are transformed, by the **output_data**, into an output format expected by the *application* component, which receives the computation result. Such an architecture could be used, for example, to implement the *word frequencies* computation illustrated in Fig. 1.

In the MapReduce system of Fig. 4, notice the presence of action ports in each component. For example, the mapper and the reducer agents have action ports connected to the *workflow* component through the action bindings **task_map**, **task_reduce_1** and **task_reduce_2**, respectively. They have four action names: a pair of alternative action names, **read_chunk** and **finish_chunk**, for signaling that, from the **collect_pairs** port, either a chunk of KV/KMV-pair can be read or there is no chunk to be read, respectively, making it possible to code the termination condition of a loop that reads an input list of chunks; **perform**, for signaling that a chunk of KV-pairs is ready to be processed (i.e. to apply map/reduce functions); and **chunk_ready**, for signaling that a new chunk of KV-pairs is available in the **feed_pairs** port. In turn, the connectors have only the **read_chunk**/**finish_chunk** and **chunk_ready** action names, distributed in their facets. For instance, shuffler has a single action port in its collector facet, carrying only the **read_chunk**/**finish_chunk** alternative action names. It is named **task_shuffler_collector_read_chunk**. Also, it has a couple of action ports in its feeder facet, **task_shuffler_feeder_read_chunk** and **task_shuffler_feeder_chunk_ready**, carrying, respectively, action names **read_chunk**/**finish_chunk** and **chunk_ready**.

The MapReduce framework is also used to exemplify contextual contracts (Section 2.5) and as case study on verification of workflows (Section 6.2).

2.3. Stakeholders

The following stakeholders work around HPC Shelf:

- The ***specialists*** (end users) use applications for specifying problems using a domain-specific interface. They wait for the execution of computational solutions built by the application for these problems, in the form of parallel computing systems. They do not handle directly with components, which are hidden behind the domain-specific abstractions of the application interface.
- The ***providers*** create and deploy applications, by designing their high-level interfaces and by programming the generation of parallel computing systems. They have skills in building computational solutions for problems in the application domain, by looking for the appropriate components and combining/orchestrating them.
- The ***developers*** write the code of component implementations, being concerned on how to tune them for better exploiting the architectural features of classes of virtual platforms. For that, they are experts in parallel computer architectures and parallel programming.
- The ***maintainers*** offer parallel computing infrastructure on top of which virtual platforms are instantiated. Through contextual contracts, they may specify the architectural features of the virtual platforms they support. Because platforms are treated as a kind of component, maintainers must register the contextual contracts of virtual platforms they support (can instantiate) in the same catalog where developers register their software components.

2.4. Architecture

The multilayer cloud architecture of HPC Shelf for servicing applications comprises the three elements in Fig. 5: Frontend, Core and Backend.

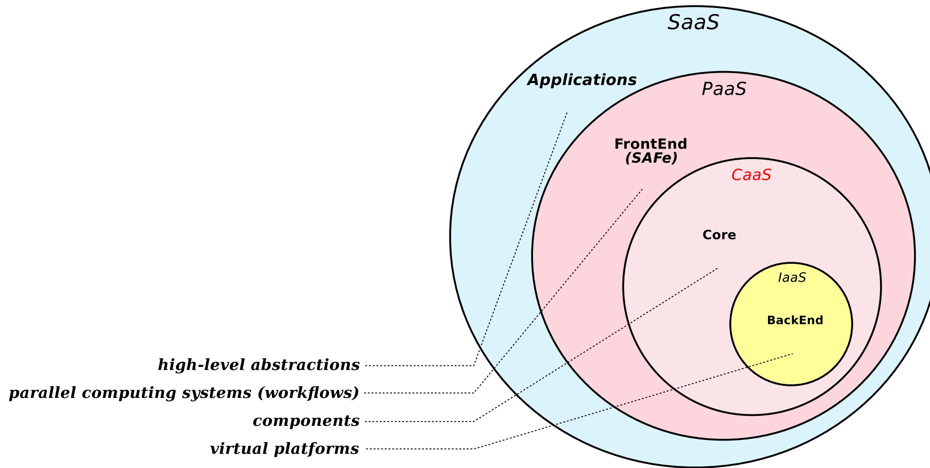

Fig. 5. The cloud architecture of HPC Shelf.

Table 1
Contextual parameters of MRComputation.

Name	Bound	Description
input_key_type	Data	the type of keys in input pairs
input_value_type	Data	the type of values in input pairs
function	Function	the custom function (e.g. map or reduce)
output_key_type	Data	the type of keys in output pairs
output_key_value	Data	the type of values in output pairs

The Frontend is SAFe (*Shelf Application Framework*) [1], a collection of classes and design patterns used by *providers* to build applications. Its current implementation is written in C#. It supports SAFeSWL as a language for specifying parallel computing systems. SAFeSWL is divided in two subsets. Through the *architectural subset*, the provider may specify which components and bindings will form the parallel computing system. In turn, using the *orchestration subset*, the provider may program its workflow, by specifying the order in which action names must be activated. Remember that a *workflow* component is responsible to execute the orchestration part of SAFeSWL code.

The Core manages the lifecycle of components, from cataloging to deployment, and implements an underlying system of *contextual contracts*. Developers and maintainers register components and their contracts through the Core. Applications access the services of the Core for resolving contextual contracts and deploying the selected components on virtual platforms.

The Backend is a service offered by each *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, without the intermediation of the Core.

2.5. Contextual contracts

HTS (Hash Type System) [12] is a type system firstly introduced by HPE (Hash Programming Environment) [8,13], the first reference implementation of the Hash component model, for the following purposes:

- The separation between specification (interface) and implementation of components, for promoting modularity and safety;
- The support of alternative implementations of a given component specification for different execution contexts, where an *execution context* is defined by the requirements of the host application and the architectural characteristics of the target parallel computing platform;
- The dynamic selection among a set of alternative component implementations according to the execution context.

Component specifications are so-called *abstract components*, whereas component implementations are so-called *concrete components*. Abstract components represent a set of components with the same interface, implementing the same concern under assumptions of distinct execution contexts.

For contextual abstraction, an abstract component declares a *contextual signature* comprising a set of *context parameters*, each one representing a placeholder for associating a particular context assumption.

In what follows, we resort to the MapReduce framework introduced in Section 2.2 for providing examples of contextual contracts. Indeed, Table 1 presents the contextual signature of an abstract component named MRComputation, specifying

Table 2
Contextual parameters of MRConnector.

Name	Bound	Description
key_type	Data	The type of keys in pairs
value_type	Data	The type of values in pairs
partition_function	Partition	The custom function for distributing keys across mappers or reducers

Table 3
Contextual contracts of Mapper and Reducer in *word frequencies*.

Parameter name	Component	Contextual bound
input_key_type	Mapper	Integer
	Reducer	String
input_value_type	Mapper	String
	Reducer	Integer
function	Mapper	CountWords[...]
	Reducer	SumValues[...]
output_key_type	Mapper Reducer	String
output_value_type	Mapper Reducer	Integer

a set of context parameters, each one with a name and a bound type. For instance, ***function*** is the name of a context parameter typed by an abstract component named Function. Since both abstract components named Mapper and Reducer are derived from MRComputation, ***function*** is used to specify the custom *map* and *reduce* functions that they will execute in particular MapReduce computations. For that, the bound of ***function*** is narrowed to the abstract components MapFunction and ReduceFunction, respectively. For that, MapFunction and ReduceFunction must be subtypes of Function.

In turn, Table 2 presents the contextual signature of MRConnector, from which the abstract components Splitter and Shuffler are derived. Besides the types of keys and values in KV/KMV-pairs, it defines a context parameter named ***partition_function***, which defines how output KV/KMV-pairs yielded after processing input KV-pairs received from **collect_pairs** bindings are distributed across **feed_pairs** ports. By configuring this parameter, users may control load balancing among mappers and reducers.

A *contextual contract* is an abstract component whose context parameters have particular execution context assumptions associated to each one of them. These assumptions are so-called *context arguments*. Indeed, the type of a context parameter is a contextual contract and a context argument is a contextual contract that is compatible with the bound type of the context parameter to which it is associated. Since contextual contracts may be interpreted as component types, so-called *instantiation types*, the compatibility relation between contextual contracts is defined as a subtype relation.

A concrete component declares the contextual contract it implements. Also, in parallel computing systems, components are specified by contextual contracts where some context parameters may be kept free, i.e. with no context argument associated to them. When a **resolve** action is activated for one of these contextual contracts, the Core triggers a resolution procedure for selecting a concrete component whose contextual contract is a subtype of it, by taking into consideration only non-free context parameters.

Table 3 presents the set of contextual arguments applied to the mapper and reducer agents of a MapReduce parallel computing system that implements the *word frequencies* example.

For that, Integer and String must be subtypes of Data, whereas CountWords and SumValues must be subtypes of MapFunction and ReduceFunction, respectively.

In the MapReduce framework, generic concrete components have been implemented for Mapper and Reducer, whose context arguments are the bounds of the context parameters. However, they could coexist with specialized versions that take advantage of knowing which data structures are used in keys and values, as well as particular map and reduce functions.

Alite. HTS has been recently extended to support context parameters representing QoS (Quality-of-Service) and cost assumptions in context contracts, receiving the name Alite [14]. Such assumptions are particularly relevant in the context of cloud computing. For that, a new kind of quantifier components has been introduced for dealing with numerically valued context parameters. Also, the resolution procedure is now composed of two stages. In the *selection stage*, the contextual contractual of a *system component* (a combination of the contracts of a computation component and its target virtual platform) is resolved by filtering all pairs of *candidate system components* that satisfy contract restrictions. In the *classification phase*, the list of candidate system components is ordered taking into account the best fulfillment of the contract requirements and the resource allocation policies of HPC Shelf. The best classified system component is selected.

3. The certification framework

For the purpose of leveraging component certification in HPC Shelf, a *certification framework* is introduced in this section. It encompasses a set of component kinds, composition rules and design patterns. They provide an environment where certification tools can be encapsulated into components to provide some level of assurance to application providers and component developers that components of parallel computing systems meet a predetermined set of requirements prior to their instantiation. Among the targeted requirements, they are included functional, non-functional and behavioral ones. However, this work is more focused on the verification of functional and behavioral properties through theorem provers and model checkers.

Certifier components constitute the main kind of components introduced by the certification framework. *Tactical components* constitute the other kind. A certifier encapsulates a *certification procedure* that can be applied to a set of *certifiable components* of a given kind, by orchestrating actions of a set of tactical components. Each tactical component encapsulates an automatic verification tool that is required for checking a set of formal properties on each certifiable component. Using this approach, a certifier can use a variety of verification tools through the set of tactical components it orchestrates.

In a parallel computing system, certifiable components are associated with one or more certifiers through *certification bindings*. Also, the same certifier may be associated to one or more certifiable components. A certifiable component must be associated with a certifier to which it is compatible through the contextual contract of the certification binding. Each certifier associated with a certifiable component may impose its own set of obligations on compatible certifiable components, such as the use of certain programming languages, design patterns, code conventions, annotations, etc.

Certifiable components have an additional **certify** action name in their lifecycle ports. When activated, each certifier associated with the certifiable component initiates a certification procedure that verifies the certifiable component against formal properties of the following kinds:

- *default properties* are predetermined for each certifier. All certifiable components associated with a certifier will be checked against them.
- *contractual properties* are also predetermined for each certifier. However, they may be configured through the contextual contracts of the certifiers, by component developers or application providers. When contextual parameters associated to a contractual property are configured by the developer of a given certifier component, the configuration is valid for any instance of the certifiable component. Otherwise, different configurations may be applied according to the needs of the application developer.
- *component properties* are defined in abstract certifiable components, so certifiable component implementations must address them.
- *ad hoc properties* are provided dynamically by the application component of the parallel computing system, through a service binding dedicated for this purpose. The service interface determines which kind of ad hoc properties are supported and how they are specified.

The certification framework also introduces a new class of stakeholders in HPC Shelf to deal with certifier and tactical components: the *certification authorities*. They are experts in applying computational methods for the certification of component requirements, including formal methods with strong mathematical foundations. Also, they are able to explore parallel programming techniques to speed up certification procedures. Certification authorities of HPC Shelf may be selected by component developers according to their reputation and transparency in the techniques and methods they use in certification procedures. They shall make publicly available the obligations imposed by certifiers to component developers. Also, for application providers, they must inform the format in which adhoc properties must be entered through adhoc properties bindings, when they exist.

3.1. Parallel certification systems

Certifier components are implemented as *parallel certification systems*, comprising the following architectural elements, as depicted in Fig. 6:

- A set of tactical components;
- A *certification-workflow* component that orchestrates the tactical ones;
- A set of bindings, connecting the tactical components to the *certification-workflow* component.

The *certification-workflow* component performs a certification procedure on the certifiable components connected to the certifier. Parallel certification systems are analogous to parallel computing systems, but aimed at certification purposes. In such an analogy, the *certification-workflow* component plays the role of the *workflow* component, also running in the memory space of SAFe. In turn, tactical components play the role of solution components. However, tactical components are, by definition, tightly coupled to the virtual platforms where they run. For this reason, they must be seen as special kinds of virtual platforms on which the proof infrastructure is installed and ready to run verification tasks.

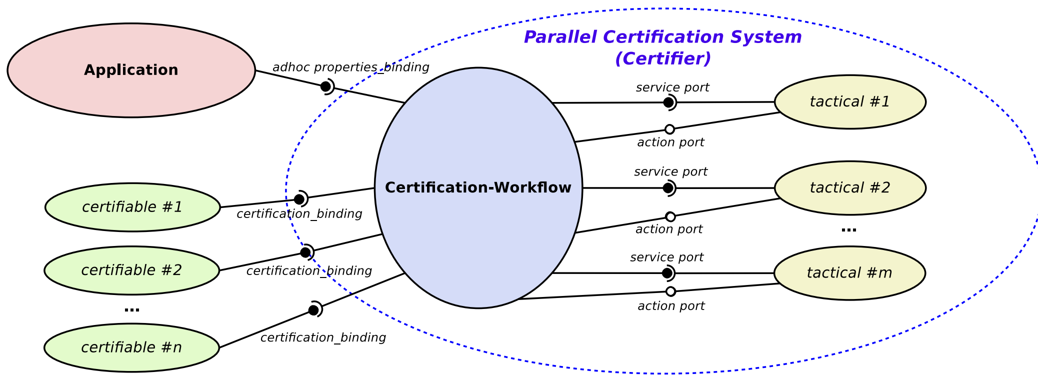

Fig. 6. The architecture of a parallel certification system (certifier component).

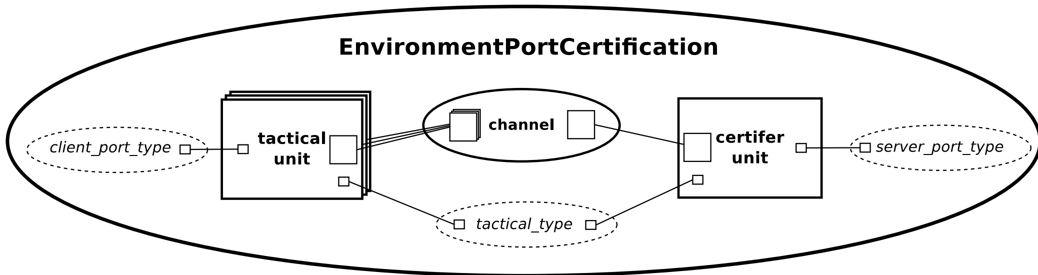

Fig. 7. Hash diagram for the service binding of tactical components.

3.2. Tactical components

As stated earlier, a tactical component encapsulates a certification infrastructure comprising one or more certification tools. Through a service port, acting as a client, it fetches input data from the *certification-workflow* component to perform certification subroutines required by the certification procedure. The nature and format of such data is freely determined by each tactical component, possibly comprising code fragments of certifiable components, as well as specification code generated by the *certification-workflow* component in a prescribed format. Thus, the tactical component may reject input data received from the *certification-workflow* component if it does not conform to its prescription.

Because tactical components are special-purpose virtual platforms, they have the full flexibility to exploit parallel processing to make memory/processing-intensive subroutines viable.

The interface of a tactical component comprises the following ports:

- A user service port through which *certification-workflow* provides operations for the following purposes, among others when it is deemed necessary:
  - receiving code fragments of certifiable components, possibly translated by *certification-workflow* into the format that it prescribes;
  - receiving specifications of formal properties to be verified;
  - monitoring the status of certification subroutines under execution;
  - informing the result of certification subroutine runs.
- An action port with action names **perform**, **conclusive** and **inconclusive**, where the latter two are alternative;
- The default lifecycle port.

Fig. 7 describes the architecture of the service binding that connects a tactical component to *certification-workflow*, whose component type is EnvironmentPortCertification. It is an indirect binding, derived from the EnvironmentBindingBaseIndirect, thus requiring two distinct units and inheriting the inner components **channel**, **server_port_type** and **client_port_type**. While the former one is used for communication between the **tactical** and **certifier** units, the latter two are qualifiers that define the server (certifier side) and client (tactical side) interface types (operation signatures) of the binding, respectively. To do so, they are typed as PortTypeServiceCertifier and PortTypeServiceTactical, respectively derived from EnvironmentPortTypeSinglePartner and EnvironmentPortTypeMultiplePartner. Note that the tactical unit is parallel, because it is designed to be a slice of the tactical component, while the certifier unit is sequential, as it is a slice of *certification-workflow*. In turn, **tactical_type** is an additional qualifier that determines the type of the tactical component. The component types of the inner components **tactical_type**, **server_port_type** and **client_port_type** determine the context signature of EnvironmentPortCertification, i.e.

$$\begin{bmatrix} \textbf{\textit{tactical_type}} = \text{TT: TacticalType} \\ \textbf{\textit{server_port_type}} = \text{S: PortTypeServiceCertifier} \\ \textbf{\textit{client_port_type}} = \text{C: PortTypeServiceTactical } [\textbf{\textit{tactical_type}} = \text{TT}] \end{bmatrix},$$

so that the implementations of EnvironmentPortCertification may be adjusted according to the type of a specific tactical component and the type of interface between the tactical component and *certification-workflow*.

The *certification-workflow* component starts a certification subroutine in a tactical component by activating **perform**. During the process, the tactical component tries to obtain code fragments and formal property specifications from its user port. Also, it may send monitoring information to *certification-workflow*, useful in the case of long running certification subroutines. When the certification subroutine terminates, either **conclusive** or **inconclusive** may be activated by the tactical component. If the certification subroutine is conclusive for all properties, the former action is activated. If not, the latter is activated. The set of situations that may prevent the certification subroutine from being conclusive includes: the occurrence of a hardware failure in the virtual platform where the tactical component resides; incompatible format in which some formal property is specified; *timeouts* reached; and so on.

In the current implementation, the contextual signature of Tactical, the component type from which specific tactical components are derived, is similar to that of EnvironmentPortCertification.

3.3. Certifier components

As stated previously, a certifiable component must be associated with one or more certifiers in a parallel computing system. In the orchestration code, an activation of the action **certify** will instantiate a parallel certification system for each certifier, which will certificate the certifiable component in parallel. Each one may be reused to certify all certifiable components associated with the same certifier, when their **certify** actions are activated.

After the certification procedure, a certifiable component is considered certified if all default, contractual, component and *ad hoc* properties have been checked by the certifier. In this case, it becomes a *certified component* with respect to the first three kinds of properties (except adhoc properties) for the applied certifier contract. To do this, a *certificate* is registered for the certifiable component through the Core services. Consequently, the certification process may be idempotent for a given certifier contract applied to a certifiable component, restricted to default, contractual and component properties. Thus, in a subsequent certification of the same certifiable component, if no *ad hoc* property is informed and the certifier contract is a supertype of some certifier contract previously applied, the certification process is no longer performed. The previous certification result is reused, making the creation of the parallel certification system no longer necessary.

As clients of certification bindings, certifiers may communicate with certifiable components even before being deployed to their virtual platforms. Through the operations they offer, the certifiers may obtain the code of certifiable components that they will verify. The response of certifiable components to such requests may be seen as a kind of reflection feature. In the current implementation, the component type of certification bindings is CerticationBinding, whose architecture is similar to EnvironmentPortCertification, with the exception that it has an inner component called **certifier_type** playing the role of **tactical_type**. So, the context signature of CerticationBinding is

$$\begin{bmatrix} \textbf{\textit{certifier_type}} = \text{TT: TacticalType} \\ \textbf{\textit{server_port_type}} = \text{S: PortTypeServiceCertifier} \\ \textbf{\textit{client_port_type}} = \text{C: PortTypeServiceTactical } [\textbf{\textit{tactical_type}} = \text{TT}] \end{bmatrix}.$$

In turn, for the purpose of receiving and configuring *ad hoc* properties, the certifiers have a user port that is commonly used to connect to a provider port of the application component. The component type of the binding responsible for connecting these ports is AdHocPropertiesBinding. Unlike EnvironmentPortCertification and CertificationBinding, it is derived from EnvironmentBindingBaseDirect, since *certification-workflow* and the application component reside in SAFe. Therefore, a **channel** component is not necessary, and there is a single interface type to define the operation signature. Thus, the context signature of AdHocPropertiesBinding is

$$\begin{bmatrix} \textbf{\textit{certifier_type}} = \text{TT: TacticalType} \\ \textbf{\textit{port_type}} = \text{S: PortTypeServiceCertifier} \end{bmatrix}.$$

Regarding the action ports, the *certification-workflow* component has ports to be connected to the action ports of its tactical components. Through these ports, the certifier may orchestrate the tactical components through the action names **perform**, **conclusive** and **inconclusive**, as already explained.

The use of multiple instances of tactical components in a certification procedure is an important feature of the certification framework. For example, such a feature may be useful when a property cannot be proven by some proof infrastructure, but can be proven by another, encapsulated in a different tactical component. In addition, in the case of multiple instances of the same tactical component, two or more instances may distribute the certification workload among them, or even overlap certification tasks, trying to reduce certification time through large-scale parallel processing techniques.

Certifiers must be derived from the abstract component Certifier, with contextual signature Certifier[***certifier_type*** = TT : CertifierType]. It is the top-level component among components of kind certifier.

4. C4: certifiers for computation components

Using the certification framework introduced in Section 3, a class of certifiers for computation components, called C4, is proposed. The name is an acronym for *Certifier Components of Computation Components*.

The units of a computation component may be viewed as processes running on different processing nodes of a virtual platform. These units can be aggregated into a *parallel unit* that represents a team of units programmed in the SPMD (Single Program Multiple Data) parallel programming pattern, so that the same code is executed on each unit. In this case, the units may run on different data partitions and synchronize by exchanging messages in a discipline akin to the MPI programming model [15]. Other variants, e.g. involving multiple parallel and singleton units running different message-passing code, are also supported by computation components.

The verification of both functional and safety properties of parallel programs based on message passing is a challenging task. For this reason, the next section provides an overview of formal verification tools that can be used in designing *tactical components* for C4 certifiers.

4.1. Tactical components for C4

The verification of computation components may resort to two different classes of methods and tools. The first class is based on *deductive program verification*, which partially automate axiomatic program verification based on some variant of the Floyd-Hoare logic. The alternative approach explores the space of reachable states of a system through *model checking*.

4.1.1. Deductive tactical components

Tactical components for deductive verification require the target component programs to be annotated with assertions in the style of the Floyd-Hoare logic or its extensions, namely, *separation logic* [16], for mutable data structures, *Owicki-Gries reasoning* [17], for shared-memory parallel programs, and *Apt's reasoning* [18], for distributed-memory components.

In such a context, a tactical component may be *purely assertional* or not. In the first case, all the properties submitted for verification must take the form of specification assertions, consisting namely of pre- and post-conditions of operations. In the second, *ad hoc* properties, i.e. properties created by the component developer and stored in the component to be verified, may be considered.

For this class of tactical components, the verification time is not proportional to the number of units in the component. However, the application of these tools to distributed-memory parallel programs, especially MPI ones, is still incipient. Actually, only ParTypes [19] can verify C/MPI programs, annotated in the syntax of VCC [20], against a high-level communication protocol stored by the certifier as an *ad hoc* property. On the other hand, thread-based programmed components, written in C or Java, can be verified against safety requirements explicitly annotated in the code (and formulated as separation logic assertions) with VeriFast [21]. Implicit properties, e.g. to ensure that a program does not access to unallocated memory locations, can also be considered. Finally, Frama-C [22] is a very expressive alternative for verifying sequential C programs having functions annotated with pre- and post-conditions.

VCC, VeriFast and Frama-C are *verification frontends*. This means they handle annotated code written in a high-level language, which is, at a latter stage, translated into an *intermediate verification language* in which the verification conditions are rephrased and checked through some automatic or interactive prover. Intermediate verification languages, such as Boogie [23] and Why3 [24], act as layers upon which verifiers are built for other languages.

Although there is a wide range of automatic provers, they can be organized into two main classes: SMT (*satisfiability modulo theories*) and ATP (*automated theorem provers*). The former determine when a first-order formula is satisfiable. Examples include Alt-Ergo [25], CVC3 [26], CVC4 [27] and Z3 [28]. ATP provers, on the other hand, implement logic inference trying to deduct a formula as a logical consequence of a set of axioms and hypotheses. Popular provers are E [29], SPASS [30] and Vampire [31].

Some provers are interactive in the sense that human intervention is required along the development of a formal proof. In such a case, they are equipped with reasonably complex interfaces for editing, searching and choosing suitable proof procedures and heuristics. They are able to deal with high-order logics in a rather expressive and versatile way, although only in a semi-automatic form. Well known examples include Coq [32] and Isabelle/HOL [33]. Tactical components to manage such tools require ports for communication with the application component, through the certifier. Using this approach, the application may either automatically interact with the tactical component or require some intervention of the specialist user to proceed the verification subroutine. In such a scenario, the user is supposedly fluent in the logic used and its representation in the tool.

In general, a tactical component for deductive verification is composed of a verification frontend, an intermediate verification language, and a prover. Other elements may appear. For example, plugins of verification frontends, such as Jessie and WP, for Frama-C, and ParTypes, for VCC. Whenever there is a single possible composition, the tactical component is named after the most specialized tool used. For example, ParTypes is composed of ParTypes, VCC, Boogie, and Z3. In other cases a composite name is used as an acronym of the tools considered. For example, JFWA is composed of Jessie, Frama-C, Why3, and Alt-Ergo.

4.1.2. Model checking tactical components

Model checking provides a powerful alternative to deductive verification tools to establish properties of MPI programs. In the context of the certification framework discussed in this article, the following tools were explored: ISP (In-situ Partial Order) [34] and CIVL (Concurrency Intermediate Verification Language) [35]. Both verify a fixed (standard), although sufficiently expressive set of safety properties. The former handles deadlock absence, assertion violations, MPI object leaks, and communication races (i.e. unexpected communication matches) in components written in C, C++ or C#, carrying MPI/OpenMP directives. CIVL, on its turn, is also able to establish functional equivalence between programs and is able to discharge verification conditions to the provers Z3, CVC3 and CVC4. In the rest of the article, CZ refers to the combination of CIVL and Z3.

4.2. Contextual contracts and architecture

It is proposed an abstract certifier so-called C4, with contextual signature

$$\text{C4}\begin{bmatrix}\textbf{\textit{certifier_type}} = CType : \textsc{CertifierType}, \\ \textbf{\textit{programming_language}} = PL : \text{PL}\textsc{Type}, \\ \textbf{\textit{message_passing_library}} = MP : \text{MP}\textsc{Type}, \\ \textbf{\textit{adhoc_properties}} = AH : \textsc{Boolean}\end{bmatrix},$$

by derivation from $\textsc{Certifier}[\textbf{\textit{certifier_ type}} = CType]$.

Thus, C4 certifiers may prescribe the host programming language on which the computation component is written, as well as the message passing library for communication between the units of the computation component. Also, certifiers can determine whether or not the *ad hoc* properties are supported. If not, the certifier does not have a port to fetch *ad hoc* properties.

From C4, two certifiers are derived to meet the proof-of-concept prototype requirements of the certification framework. They are:

- C4MPISimple, dedicated to the model checking verification of components implemented by using C and MPI, not requiring annotations in programs and not supporting component and *ad hoc* properties;
- C4MPIComplex, dedicated to the verification of a richer set of formal properties (default, component and *ad hoc* ones) through various tactical components encapsulating model checkers and deductive provers.

C4MPISimple extends C4 by closing all context parameters, i.e. by

$$\text{C4}\begin{bmatrix}\textbf{\textit{certifier_type}} = \text{C4MPI}\textsc{SimpleType}, \\ \textbf{\textit{programming_language}} = \text{C}, \\ \textbf{\textit{message_passing_library}} = \text{MPI}, \\ \textbf{\textit{adhoc_properties}} = \textsc{False}\end{bmatrix},$$

so that it can verify C/MPI programs and ignores *ad hoc* properties. It has a single tactical component, based on ISP, described later.

In turn, C4MPIComplex extends C4 by

$$\text{C4}\begin{bmatrix}\textbf{\textit{certifier_type}} = \text{C4MPI}\textsc{ComplexType}, \\ \textbf{\textit{programming_language}} = \text{C}, \\ \textbf{\textit{message_passing_library}} = \text{MPI}, \\ \textbf{\textit{adhoc_properties}} = \textsc{True}\end{bmatrix},$$

introducing *ad hoc* properties. However, it adds its own context parameters:

$$\text{C4MPI}\textsc{Complex}\begin{bmatrix}\textbf{\textit{separation_logic}} = S : \textsc{Boolean}, \\ \textbf{\textit{floating_point_operations}} = F : \textsc{Boolean}, \\ \textbf{\textit{existential_quantifier}} = E : \textsc{Boolean}\end{bmatrix}.$$

Such parameters are used by developers for providing some information to the certifier that may be relevant to guide the verification process:

- ***separation_logic*** states whether annotations with separation logic assertions must be considered;
- ***floating_point_operations*** indicates whether the certifier must verify floating point operations;
- ***existential_quantifier*** indicates whether the assertions contain existential quantifiers.[4]

[4] Note that SMT solvers usually have difficulties at guessing witnesses for existentially quantified variables [36]. In this case, if the certifier orchestrates a set of SMT solvers and ATP provers, it is advisable that the latter are applied in first place.

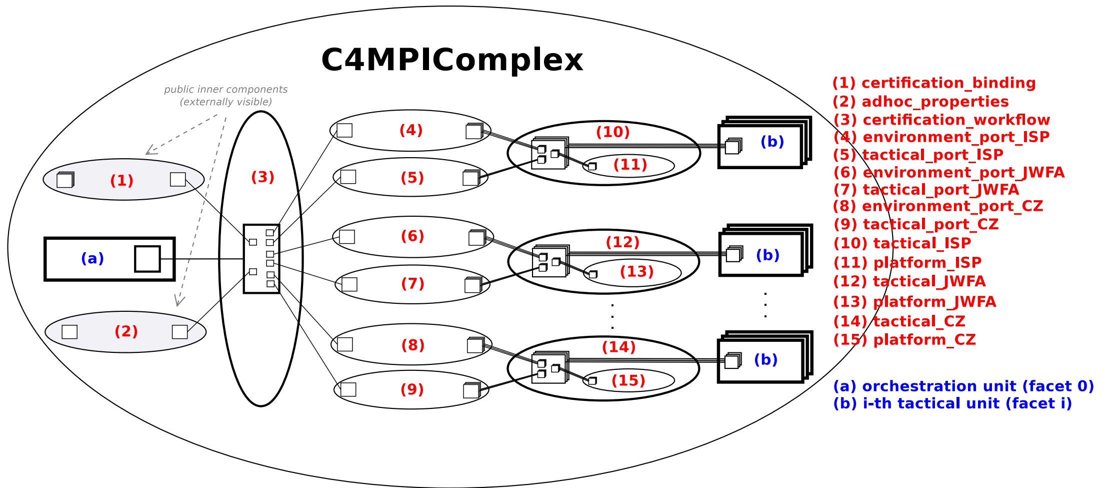

Fig. 8. A Hash diagram for C4MPIComplex (ignoring qualifiers).

The architecture of C4MPIComplex is represented in Fig. 8, using a Hash diagram. It follows the general architecture diagram for parallel certification systems presented in Fig. 6. Therefore, it comprises a set of tactical components, each connected to the *certification-workflow* component by means of an environment binding and an action binding. In addition, a pair of environment bindings (the mandatory *certification binding* and the optional binding of *adhoc properties*) are responsible to connect the certifier to the parallel computing systems (through the certifiable and application components, respectively). There is a singleton unit, which runs on SAFe, hosting the single *certification-workflow* unit, and three parallel units, each one running in a tactical component. Because they run on distinct virtual platforms, C4MPIComplex comprises four facets.

The tactical components of C4MPIComplex are ISP, JWFA and CZ, whose abstract components restrict the bounds of the context parameters of Tactical to define the interface types through which they talk to the *certification-workflow* component. For example, ISP has the signature

$$\text{ISP}\left[\begin{array}{l}\boldsymbol{tactical_type} = TT : \text{TacticalTypeISP}, \\ \boldsymbol{server_port_type} = S : \text{PortTypeISP}, \\ \boldsymbol{client_port_type} = C : \text{PortTypeISP}\end{array}\right],$$

by extending Tactical with

$$\text{Tactical}[\boldsymbol{tactical_type} = TT, \boldsymbol{server_port_type} = S, \boldsymbol{client_port_type} = C]$$

There is a single concrete component for each abstract component ISP, JWFA, and CZ, closing contextual parameters with their bounds (generic implementations). For example, the concrete component JFWAImpl implements the contextual contract

$$\text{JWFA}\left[\begin{array}{l}\boldsymbol{tactical_type} = \text{TacticalTypeJWFA}, \\ \boldsymbol{server_port_type} = \text{PortTypeJWFA}, \\ \boldsymbol{client_port_type} = \text{PortTypeJWFA}\end{array}\right].$$

In the current implementation, the concrete components ISPImpl, LWFAImpl, and CZImpl have been created as images of virtual machines hosted on the EC2 IaaS service of Amazon.[5]

C4MPISimple is a simpler version of C4MPIComplex, having only the tactical component ISP. In fact, Table 4 delineates the properties that both are able to verify by emphasizing which tactical components are involved in the verification. The reader may conclude that C4MPISimple is a subset of C4MPIComplex in terms of verification expressiveness.

5. SWC2: certifiers for workflow components

As described in Section 2, in the architecture of a parallel computing system, a singleton component called *workflow* represents the orchestration engine. In the current implementation of HPC Shelf, it may be implemented in two ways:

- using a host programming language (currently, C#), by activating actions using the `ITaskBinding` interface of action ports (Listing 1);

[5] https://aws.amazon.com/ec2.

Table 4
Properties of C4MPISimple and C4MPIComplex.

Certifier	Tactical	Properties			
		Default	Contractual	Component	Ad hoc
C4MPISimple	ISP	Deadlock Object Leaks Communication Races Irrelevant Barriers	No	No	No
C4MPIComplex	ISP	Deadlock Object Leaks Communication Races Irrelevant Barriers			
	JWFA			Annotations in Frama-C ACSL[a] syntax	
	CZ	Out-of-bound Arrays Memory Leaks Pointer Arithmetic	No	Annotations in CIVL style for defining a sequence of refinement layers that will take the original program to an abstract, then verifiable, parallel program	External parallel program for functional comparison with a component program

[a] https://frama-c.com/acsl.html.

- using SAFeSWL, a specific-purpose scientific workflow language for orchestrating parallel components, driving the execution of computational and synchronization actions of solution components.

This section introduces a certifier for *workflow* components, designated by SWC2 (*Scientific Workflow Certifier Component*). It may verify SAFeSWL orchestrations by checking a set of behavioral properties, currently using a single proofing infrastructure based on the mCRL2 tool.

5.1. Notes about SAFeSWL workflows

SAFeSWL typically specifies workflows of coarse-grained component systems, due to the sort of algorithms it has been designed to 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 workflows. Thus, orchestration languages aimed at the creation of this kind of workflows, such as SAFeSWL, generally offer few primitive constructors and combinators to express action activation (synchronous and asynchronous), sequencing, branching, iteration and parallelism.

Like in most scientific workflows management systems, such as Askalon [37], BPEL Sedna [38], Kepler [39], Pegasus [40], Taverna [41] and Triana [42], SAFeSWL workflows are usually represented by components and execution dependencies among them, usually adopting abstract descriptions of components and abstracting away from the computing platforms on which they run. Thus, they only specify interfaces that expose a set of 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, making it relevant to ensure that the activation of computational actions of components is made after their effective resolution. Also, resolution procedures may find out which virtual platform best fits the requirements of component implementations, making it interesting to verify statically whether the computational actions of components are always activated after resolution of their host virtual platforms.

In order to minimize the waste of computational resources, virtual platforms should be instantiated only when the components that such platforms host are strictly necessary, as well as released when they are no longer needed. This is the primary motivation for supporting a component lifecycle control mechanism by using the actions **resolve**, **deploy**, **instantiate**, **run**, and **release** described in Section 2.1. The consistency of the order of activation of these actions may be statically checked, so that the lifecycle follows its natural order.

The SAFeSWL orchestration code of the *workflow* component may be seen as a behavior expression representing a protocol for exogenous activation of computational, communication and synchronization actions of solution components in a parallel computing system. However, SWC2 certifiers also need to take into account the protocol of endogenous action activations performed by these solution components, called *internal workflows*. Thus, the verification procedure over the workflow of a parallel computing system is performed in a workflow generated by a composition of the main orchestration code, specified by the *workflow* component, with the orchestration codes of internal workflows, for each solution component. By taking internal workflows into account, SWC2 certifiers may refine the verification process, making possible to check a richer set of useful properties. However, since solution components are programmed in a general-purpose programming language (currently, C#), the code of internal workflows are specified by abstract components, using SAFeSWL, making their obedience an obligation of component developers when implementing concrete components.

5.2. Architecture and contextual contracts

SWC2 prescribes two default properties:

- Deadlock absence;
- Obedience to the protocol in which lifecycle actions must be activated, for each component, presented in Section 2.1.

Also, it has a contractual property that specifies whether *absence of infinite loop* must be verified. Therefore, the abstract component SWC2 has the following contextual signature:

$$\text{SWC2}\begin{bmatrix}\textbf{\textit{certifier_type}} = CType : \textsc{CertifierType}, \\ \textbf{\textit{infinite_loop_absence}} = I : \textsc{Boolean}, \\ \textbf{\textit{adhoc_properties}} = A : \textsc{Boolean}\end{bmatrix}$$

Thus, SWC2 verifiers decide whether they accept, or not, *ad hoc* properties. Also, they may include other default and contractual properties, as well as component properties in some prescribed format.

By derivation from SWC2, a certifier based on the mCRL2[6] tool has been proposed. It supports all the default properties prescribed for SWC2 certifiers, as well as *ad hoc* properties. Also, it keeps the verification of infinite loop absence as a contractual property. Thus, its contextual signature is:

$$\textsc{mCRL2Certifier}\big[\textbf{\textit{infinite_loop_absence}} = I : \textsc{Boolean}\big],$$

$$\text{by extending SWC2}\begin{bmatrix}\textbf{\textit{certifier_type}} = \textsc{mCRL2Type}, \\ \textbf{\textit{infinite_loop_absence}} = I, \\ \textbf{\textit{adhoc_properties}} = \textsc{True}\end{bmatrix}$$

mCRL2Certifier has a single instance of the mCRL2Tactics tactical component, thus requiring a single virtual platform. The abstract component mCRL2Tactics has the signature

$$\textsc{mCRL2Tactics}\begin{bmatrix}\textbf{\textit{server_port_type}} = S : \textsc{PortType_mCRL2}, \\ \textbf{\textit{client_port_type}} = C : \textsc{PortType_mCRL2}, \\ \textbf{\textit{version}} = V : \textsc{Integer}\end{bmatrix},$$

$$\text{by extending }\textsc{Tactical}\begin{bmatrix}\textbf{\textit{tactical_type}} = \textsc{mCRL2Tactics}, \\ \textbf{\textit{server_port_type}} = S, \\ \textbf{\textit{client_port_type}} = C\end{bmatrix},$$

being implemented by the concrete component mCRL2TacticsImpl, for the following contextual contract:

$$\textsc{mCRL2Tactics}\begin{bmatrix}\textbf{\textit{client_port_type}} = \textsc{PortType_mCRL2}, \\ \textbf{\textit{server_port_type}} = \textsc{PortType_mCRL2}, \\ \textbf{\textit{version}} = 201409\end{bmatrix}.$$

There are two concrete components of mCRL2Certifier. They are called mCRL2CertifierImpl1 and mCRL2CertifierImpl2, having the context parameter ***life_cycle_verification*** assigned to True and False, respectively. However, a single implementation could exist in an alternative design that could check the current value of ***life_cycle_verification*** through a service port specifically devoted to this purpose. However, this feature is not yet supported in our prototype implementation of HPC Shelf.

The mCRL2 toolkit does not support distributed-memory parallelism. Thus, it cannot distribute computations of a verification procedure between the nodes of the tactical component. However, mCRL2Certifier certifiers may be able to exploit parallelism by initiating different verification tasks on distinct processing nodes of the tactical component. In an alternative design, it would be also possible to have multiple instances of mCRL2Tactics.

5.3. 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 [43,44] was chosen here to support workflow verification. System behaviors in mCRL2 are specified in a process algebra reminiscent of ACP [45]. Processes are built from a set of user-declared actions

[6] https://www.mcrl2.org.

$$T ::= L \mid G \mid T_1; T_2 \mid T_1 || T_2 \mid \texttt{repeat}\ T \qquad \text{(task)}$$

$$L ::= act \mid \texttt{break} \mid \texttt{continue} \mid \texttt{start}(h, act) \mid \texttt{wait}(h) \mid \texttt{cancel}(h) \qquad \text{(literal)}$$

$$G ::= act{\downarrow}T \mid act{\downarrow}T + G \qquad \text{(guarded tasks)}$$

Fig. 9. Formal grammar of the orchestration subset of SAFeSWL.

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

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

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

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

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

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

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

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. 10. Operational semantics of the orchestration subset of SAFeSWL.

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 [46]; 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 [47] 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.

5.3.1. The translation process

The translation process follows directly the operational rules (Fig. 10) defined for an abstract grammar of a formal specification of the orchestration subset of SAFeSWL (Fig. 9).

Let W be the workflow component of a parallel computing system. In the grammar of Fig. 9, c ranges over component identifiers, h ranges over naturals and $act \in Act_W$. For each component, it is assumed a minimal set of workflow actions, including the life cycle ones ($\{\texttt{resolve}_c, \texttt{deploy}_c, \texttt{instantiate}_c, \texttt{run}_c, \texttt{release}_c\} \subseteq Act_W$).

The semantics of W consists of a task T_W, given by the rules in Fig. 10, and initial state $\langle T_W, \texttt{stop}, \emptyset, \emptyset, \emptyset, \emptyset\rangle$. The symbol stop denotes task completion. Each execution state is 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 of finished asynchronous actions.

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

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.

Now, it is possible to 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.

5.4. Specifying and proving default properties in mCRL2

The first default property is deadlock absence, specified as

$$\mathsf{DA}: [\mathsf{true}*]\langle\mathsf{true}\rangle\mathsf{true},$$

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. Infinite loop absence (ILA) may be checked 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 formula

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

The remaining properties express lifecycle restrictions in terms of precedence relations specified by formulas like

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

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 lifecycle actions using a similar pattern, such as:

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

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

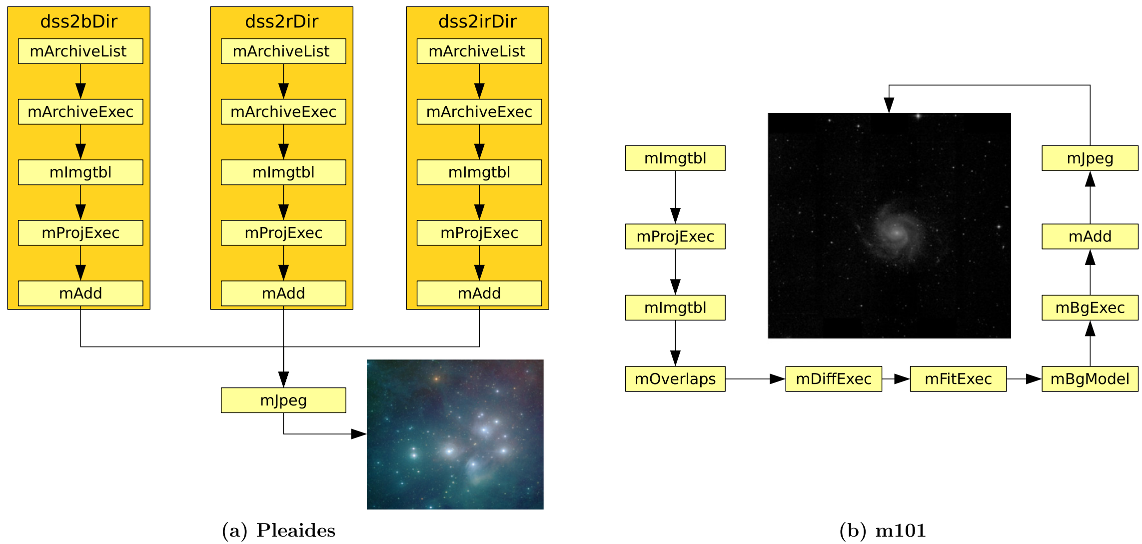

Fig. 11. Two examples of Montage workflows.

6. Case studies

In this section, three case studies demonstrate the certification framework of HPC Shelf, as well as the use of C4 and SWC2 certifiers.

Performance evaluation experiments have been performed to evidence that the cost of component certification may be reasonable and will not make it impractical to run a parallel computing system with some certifiable components. In order to isolate the pure verification times from the overheads of parallel certification system deployment, the sequential times have been calculated outside the HPC Shelf framework, through direct sequential calls to verification engines.

The experiments have been performed by instantiating tactical components, as well as their associated proof infrastructures, in a cluster installed at the Computer Science Research Laboratory (LIA) of Federal University of Ceará, comprising 8 nodes equipped with 64 GB of memory serving a pair of Intel Xeon E5-2650 V3 processors having 10 cores each.

6.1. Montage

Montage (Mosaic Astronomic Engine) [48] is an astrophotography software toolkit for the composition of astronomical mosaics (sets of images of a specific area in the sky). It preserves the position of the original input images. In fact, Montage is an alternative to the inability of astronomical cameras to handle very large images. It is often used as a case study in the evaluation of scientific workflow management systems (SWfMS).

Using Montage, one can orchestrate a set of independent, self-executing components to get a mosaic of images. Such components receive a set of files and some arguments as input. After performing the computation, they display a set of files (possibly images) that can be read by other components in the next steps of the workflow. In HPC Shelf, Montage components have been encapsulated as simple computation components having only the standard lifecycle action port. When the action **run** of one of these components is activated, the main component calculation, which is written in C, is executed.

Montage components are now illustrated using an existing workflow that generates a mosaic for the Pleiades star cluster,[7] depicted in Fig. 11(a). It has the following components:

- mArchiveList retrieves a list of images that overlaps a given sky coordinate from IRSA (Infrared Science Archive)[8] and stores them into a file whose name is also given as input;
- mArchiveExec retrieves each image listed in a file given as input from the server and stores them into the current directory, in the FITS format (Flexible Image Transport System)[9];
- mImgtbl generates, in a table, metadata information for images contained in a directory;
- mProj reprojects images from a directory by using information contained in the metadata table generated by mImgtbl;
- mAdd joins all images previously corrected into the final image;

[7] http://montage.ipac.caltech.edu/docs/pleiades_tutorial.html.
[8] http://irsa.ipac.caltech.edu/ibe.
[9] http://fits.gsfc.nasa.gov.

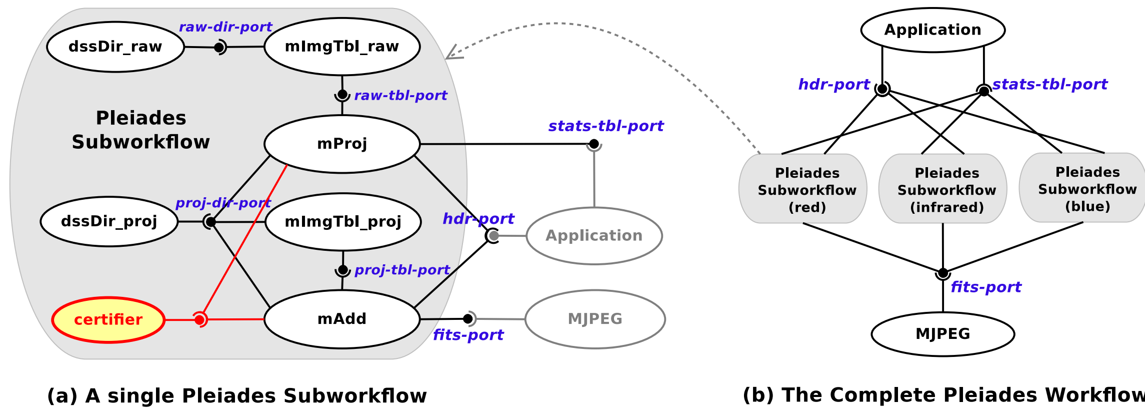

Fig. 12. The Hash architecture of Pleiades workflow.

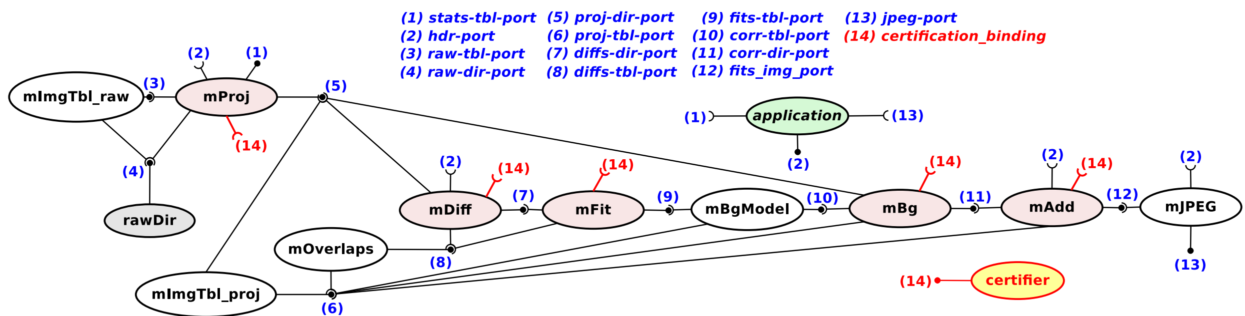

Fig. 13. The Hash architecture of M101 workflow.

- mJpeg generates the final mosaic image, in JPEG format, by combining the images of each color band in the last step of the workflow.

The workflow maintains three versions of each image, which overlap the center of the Pleiades cluster, each corresponding to a different color band: *red*, *infrared* and *blue*. They are stored, respectively, in the image repositories represented by the data source components dss2rDir, dss2irDir and dss2bDir. A separate, independent, subworkflow processes each color band in parallel. When they are finished, the resulting three images, for each color band, are overlapped by the mJPEG component to generate the final JPEG image.

Fig. 12 depicts the architecture of a parallel computing system for the Pleiades workflow. Fig. 12(a) presents the architecture of a single subworkflow. In turn, Fig. 12(b) depicts the overall architecture, including the three subworkflow instances. For simplification, the *workflow* component is omitted, but it must be assumed that components **mImgTbl_raw**, **mImgTbl_proj**, **mProj**, and **mAdd**, respectively typed by contextual contracts of MImgTbl, MProj, and MAdd, have action ports connected to it. Also, notice the presence of the **certifier** component (for each subworkflow), aimed at certifying **mProj** and **mAdd** in each subworkflow.

6.1.1. Certifying Montage components

The processing time of Montage workflows is dominated by its parallel components, since their associated sequential components perform the most critical computational tasks. Montage components that have parallel versions are mProj, mAdd, mBg, mDiff and mFit. In HPC Shelf, each has a single concrete component consisting of a single parallel unit that runs in the SPMD style. Since Pleiades uses instances of only the two previous components, this case study will use the Montage workflow that builds a mosaic of the galaxy m101,[10] illustrated in Fig. 11(b). Its corresponding parallel computing system is represented in Fig. 13. It is a sequential workflow that includes instances of all Montage components that interest to this experiment, i.e. MProj, MAdd, MBg, MDiff and MFit.

This case study demonstrates the certification of components **mProj**, **mAdd**, **mBg**, **mDiff** and **mFit** of the M101 parallel computing system, by using C4MPISimple. To do this, the application provider must configure certification bindings between these component instances and one or more instances of C4MPISimple. For M101, as shown in Fig. 13, a single C4MPISimple instance is sufficient, named **certifier**, since the certifier contract is the same for all the certifiable components.

10 http://montage.ipac.caltech.edu/docs/m101tutorial.html.

```
0  <sequence>
1  <parallel>
2  <sequence>
3  <invoke action="resolve"     id_port="mImgTbl_raw"/>
4  <invoke action="deploy"      id_port="mImgTbl_raw"/>
5  <invoke action="instantiate" id_port="mImgTbl_raw"/>
6  </sequence>
7  <sequence>
8  <invoke action="resolve"     id_port="mImgTbl_proj"/>
9  <invoke action="deploy"      id_port="mImgTbl_proj"/>
10 <invoke action="instantiate" id_port="mImgTbl_proj"/>
11 </sequence>
12 <sequence>
13 <invoke action="resolve"     id_port="mBgModel"/>
14 <invoke action="deploy"      id_port="mBgModel"/>
15 <invoke action="instantiate" id_port="mBgModel"/>
16 </sequence>
17 <sequence>
18 <invoke action="resolve"     id_port="mOverlaps"/>
19 <invoke action="deploy"      id_port="mOverlaps"/>
20 <invoke action="instantiate" id_port="mOverlaps"/>
21 </sequence>
22 <sequence>
23 <invoke action="resolve"     id_port="mJPEG"/>
24 <invoke action="deploy"      id_port="mJPEG"/>
25 <invoke action="instantiate" id_port="mJPEG"/>
26 </sequence>
27 <sequence>
28 <invoke action="resolve"     id_port="mAdd"/>
29 <invoke action="certify"     id_port="mAdd"/>
30 <invoke action="deploy"      id_port="mAdd"/>
31 <invoke action="instantiate" id_port="mAdd"/>
32 </sequence>
33 <sequence>
34 <invoke action="resolve"     id_port="mProj"/>
35 <invoke action="certify"     id_port="mProj"/>
36 <invoke action="deploy"      id_port="mProj"/>
37 <invoke action="instantiate" id_port="mProj"/>
38 </sequence>
39 <sequence>
40 <invoke action="resolve"     id_port="mBg"/>
41 <invoke action="certify"     id_port="mBg"/>
42 <invoke action="deploy"      id_port="mBg"/>
43 <invoke action="instantiate" id_port="mBg"/>
44 </sequence>
45 <sequence>
46 <invoke action="resolve"     id_port="mDiff"/>
47 <invoke action="certify"     id_port="mDiff"/>
48 <invoke action="deploy"      id_port="mDiff"/>
49 <invoke action="instantiate" id_port="mDiff"/>
50 </sequence>
51 <sequence>
52 <invoke action="resolve"     id_port="mFit"/>
53 <invoke action="certify"     id_port="mFit"/>
54 <invoke action="deploy"      id_port="mFit"/>
55 <invoke action="instantiate" id_port="mFit"/>
56 </sequence>
57 </parallel>
58 <invoke action="run" id_port="mImgTbl_raw"/>
59 <invoke action="run" id_port="mProj"/>
60 <invoke action="run" id_port="mmImgTbl_proj"/>
61 <invoke action="run" id_port="mOverlaps"/>
62 <invoke action="run" id_port="mDiff"/>
63 <invoke action="run" id_port="mFit"/>
64 <invoke action="run" id_port="mBgModel"/>
65 <invoke action="run" id_port="mBg"/>
66 <invoke action="run" id_port="mAdd"/>
67 <invoke action="run" id_port="mJPEG"/>
68 </sequence>
```

Fig. 14. SAFeSWL code of M101 parallel computing system.

Fig. 14 presents the SAFeSWL orchestration code of the M101 workflow, where the **certify** action is activated in parallel for all certifiable components. However, since there is only one instance of **certifier** to run the certification procedures, they will be serialized.

In this experiment, the C4MPISimple certifier uses a single tactical component (instance of ISPImpl) with 5 units (each placed in a processing node).

The **certify** action activation concludes successfully for all certifiable components in M101, except **mBg**. Thus, according to the default properties of C4MPISimple, they are free of deadlocks, object leaks, communication races and irrelevant barriers. For **mBg**, ISP detects a possible interleaving that can lead to receiving an empty buffer in an invocation of `MPI_Allreduce`.

Fig. 15 reports execution times for this certification case study, by varying the number of processing nodes and cores per node involved in the execution of the tactical component ISP.[11] For simplicity, the sequential time has been calculated outside HPC Shelf through sequential calls to the verification tool, and collected through a shell script program. This is a

[11] The maximum number of 5 nodes was chosen because ISP is able to check the properties of a single component of each a time.

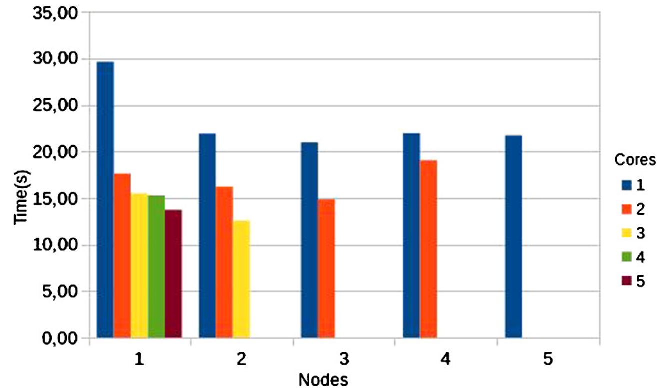

Fig. 15. Execution times in the certification of M101 components.

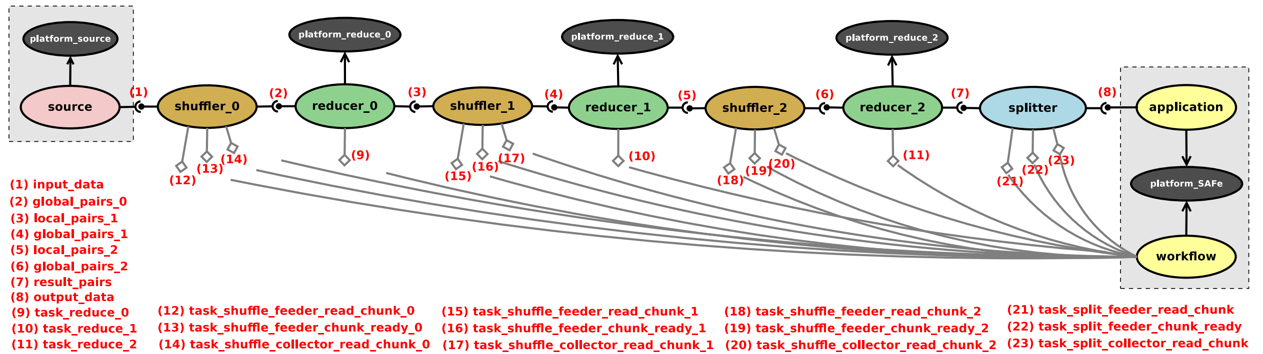

Fig. 16. Architecture of a non-iterative MapReduce system with three stages.

reasonable assumption since the interest, in this performance evaluation, is to isolate the impact of parallel certification system deployment in the overall certification time.

Parallel execution has been justified for all cases. In some cases, the time needed for a similar certification in a purely sequential scenario has been reduced by a factor of two. This gain is likely to become even more evident if more programs are considered for certification within a component, or when the certifier must orchestrate a greater number of tactical components.

6.2. MapReduce workflows

This case study demonstrates the use of the mCRL2Certifier certifier to prove properties about two particular workflows of MapReduce parallel computing systems (Section 2.2), whose architectures are depicted in Figs. 16 and 17. The first one is a non-iterative system with three reducing stages, while the latter one is an iterative system with a single reducing stage. They do not present mapping stages. These systems have been adapted from case studies on the implementation of graph calculations, respectively *triangle enumeration* and *single-source shortest path* (SSSP), by using Gust, a framework for processing big graphs derived from the MapReduce framework [11].

6.2.1. The non-iterative system with three stages

In the non-iterative system with three stages (Fig. 16), each stage comprises a pair of a shuffler and a reducer, that is $\langle \mathbf{shuffler}_i, \mathbf{reducer}_i \rangle$, for $i \in \{0, 1, 2\}$. They are the intermediate stages of a pipeline. The source of the pipeline is the data source component called **source**, while its sink is the *application* component. It is required an intermediate splitter (**splitter**) for communication between **reducer_2** and *application*, since they are placed in distinct virtual platforms.

The workflow of the non-iterative system initially performs the lifecycle action activation sequence (**resolve**, **deploy**, **instantiate**, and **run**) for all components, because, in a pipeline pattern, they are required from the beginning of the computation. The components are divided into three groups: virtual platforms; computations and connectors; and bindings. The components in the same group are instantiated in parallel. First, all virtual platforms are instantiated. Then, computations and connectors that will be placed on these virtual platforms. Finally, all the bindings that connect the computations and

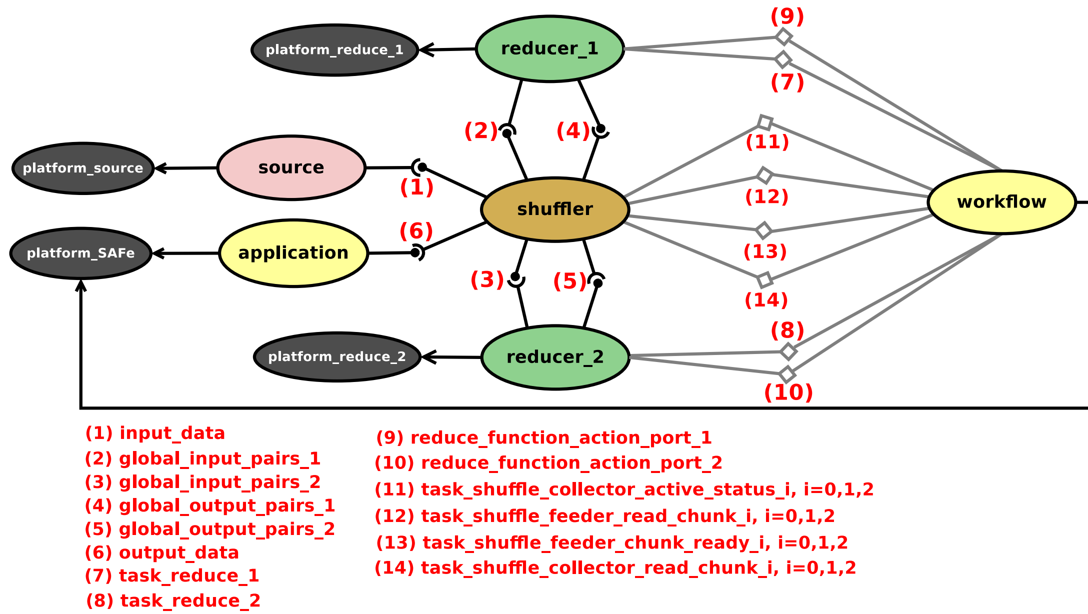

Fig. 17. Architecture of an iterative MapReduce system.

connectors. The SAFeSWL activation sequence that instantiates each component is described below, where *component_id* represents the component identifier[12]:

```
0  <sequence>
1  <invoke port=component_id action="resolve"/>
2  <invoke port=component_id action="deploy"/>
3  <invoke port=component_id action="instantiate"/>
4  </sequence>
```

After all the components have been instantiated, the action **run** is activated for the shufflers and reducers, and the orchestration of computational actions is started. It consists of four iterations, one for each intermediate stage and one for the sinking stage of the pipeline. Iterations are run in parallel because the pipeline pattern assumes that all stages are active at the same time. For example, the iteration code corresponding to the i-th intermediate stage of the pipeline, where $i = \{0, 1, 2\}$, is described below:

```
0  <sequence>
1  <iterate port="task_shuffle_i_collector_read_chunk" until="FINISH_CHUNK" loop="READ_CHUNK">
2  <sequence>
3  <invoke port="task_shuffle_i_feeder_read_chunk" action="READ_CHUNK" />
4  <invoke port="task_shuffle_i_feeder_chunk_ready" action="CHUNK_READY" />
5  <invoke port="task_reduce_i" action="READ_CHUNK" />
6  <invoke port="task_reduce_i" action="PERFORM" />
7  </sequence>
8  </iterate>
9  <invoke port="task_shuffle_i_feeder_read_chunk" action="FINISH_CHUNK" />
10 <invoke port="task_reduce_i" action="FINISH_CHUNK" />
11 <invoke port="task_reduce_i" action="CHUNK_READY" />
12 </sequence>
```

After all the iterations are terminated, the parallel activation completes and all components are released.

6.2.2. The iterative system with a single stage

In the iterative workflow (Fig. 17), the single stage consists of a shuffler and a pair of parallel reducers. Component instantiations and releasings follow the same pattern used in the non-iterative workflow except that **output_data** is instantiated only after the main iteration, when the output pairs are available.

In the first iteration, **shuffler** reads input pairs from the data source and distributes them between **reducer_1** and **reducer_2**. Then, in the next iterations, the shuffler receives the output pairs of the two reducers, groups and redistributes them between the reducer units. The termination condition of the main iteration is detected by an alternate activation of action names **continue** and **terminate** in either the **reduce_function_action_port_1** or **reduce_function_action_port_2** port, since the termination condition, which depends on the computation being executed, is verified by the reduction function performed by the reducers. In the body of the loop, there is a code similar to the code presented earlier for reading a list of

[12] Remember that, by default, the name of the lifecycle action port of a component is the name of the component itself. This is a simple “syntactic sugar”.

pairs in a pipeline stage of the non-iterative system. However, in the iterative system, it involves the two shuffler collector facets associated with the two parallel reducers. This is:

```
<sequence>
<iterate port="task_shuffle_collector_read_chunk_1"  loop="READ_CHUNK" until="FINISH_CHUNK">
<sequence>
<invoke port="task_shuffle_collector_read_chunk_2" action="READ_CHUNK" />
<parallel>
<sequence>
<invoke port="task_shuffle_feeder_read_chunk_1"  action="READ_CHUNK" />
<invoke port="task_shuffle_feeder_chunk_ready_1" action="CHUNK_READY" />
<invoke port="task_reduce_1"                     action="READ_CHUNK" />
<invoke port="task_reduce_1"                     action="PERFORM" />
</sequence>
<sequence>
<invoke port="task_shuffle_feeder_read_chunk_2"  action="READ_CHUNK" />
<invoke port="task_shuffle_feeder_chunk_ready_2" action="CHUNK_READY" />
<invoke port="task_reduce_2"                     action="READ_CHUNK" />
<invoke port="task_reduce_2"                     action="PERFORM" />
</sequence>
</parallel>
</sequence>
</iterate>

<invoke port="task_shuffle_collector_read_chunk_2" action="FINISH_CHUNK" />

<parallel>
<invoke port="task_shuffle_feeder_read_chunk_0" action="FINISH_CHUNK" />
<invoke port="task_shuffle_feeder_read_chunk_1" action="FINISH_CHUNK" />
<invoke port="task_shuffle_feeder_read_chunk_2" action="FINISH_CHUNK" />
<invoke port="task_reduce_1"                    action="FINISH_CHUNK" />
<invoke port="task_reduce_2"                    action="FINISH_CHUNK" />
</parallel>

<parallel>
<invoke port="task_reduce_1" action="CHUNK_READY" />
<invoke port="task_reduce_2" action="CHUNK_READY" />
</parallel>
<sequence>
```

The **shuffler** connector plays a central role in the computation. It has three collector facets, one to receive input pairs of **source** in the first iteration and two to receive intermediate pairs of **reducer_1** and **reducer_2** in the subsequent iterations, and three feeder facets, two to send intermediate pairs to **reducer_1** and **reducer_2**, respectively, and a third one to send output pairs to the *application* component in the last iteration. In the internal communication of pairs of the collector facets to the feeder facets, through the channel inner component of **shuffler**, the feeder facets may receive pairs of any of the collector facets in a non-deterministic manner. To determine the end of the list of pairs, the feeder facets should receive an end marker from each collector facet. However, in the first iteration of the workflow, there is no pair to be received from **reducer_1** and **reducer_2**. In turn, in the next iterations, there is no pair to be received from **source**. So, in both cases, how is it not possible to determine the end of the input list of pairs? The solution implemented to this issue is the inclusion of an additional action port in each collector facets of **shuffler**, which are named **task_shuffle_collector_active_status_*i***, for $i \in \{0, 1, 2\}$, in Fig. 17. They have a couple of independent action names **change_status_begin** and **change_status_end**, and a couple of alternative action names **inactive** and **active**. Before the first iteration, the following code is executed to keep it enabled only the collector facet that communicates with **source**, assuming that all collector facets are enabled, by default, in the beginning of the computation:

```
<parallel>
<sequence>
<invoke port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_BEGIN" />
<invoke port="task_shuffle_collector_active_status_1" action="INACTIVE" />
<invoke port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_END" />
</sequence>
<sequence>
<invoke port="task_shuffle_collector_active_status_2" action="CHANGE_STATUS_BEGIN" />
<invoke port="task_shuffle_collector_active_status_2" action="ACTIVE" />
<invoke port="task_shuffle_collector_active_status_2" action="CHANGE_STATUS_END" />
</sequence>
</parallel>
```

In turn, before the next iterations, the following code is executed to enable the collector facets that receive pairs of the reducers and disable the collector facet that receive pairs of source:

```
<parallel>
<sequence>
<invoke port="task_shuffle_collector_active_status_0" action="CHANGE_STATUS_BEGIN" />
<invoke port="task_shuffle_collector_active_status_0" action="INACTIVE" />
<invoke port="task_shuffle_collector_active_status_0" action="CHANGE_STATUS_END" />
</sequence>
<sequence>
<invoke port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_BEGIN" />
<invoke port="task_shuffle_collector_active_status_1" action="ACTIVE" />
<invoke port="task_shuffle_collector_active_status_1" action="CHANGE_STATUS_END" />
</sequence>
<sequence>
<invoke port="task_shuffle_collector_active_status_2" action="CHANGE_STATUS_BEGIN" />
<invoke port="task_shuffle_collector_active_status_2" action="ACTIVE" />
<invoke port="task_shuffle_collector_active_status_2" action="CHANGE_STATUS_END" />
</sequence>
</parallel>
```

6.2.3. Internal workflows of MapReduce components

As explained in the end of Section 5.1, SWC2 certifiers require that abstract components of kinds computation and connector provide an orchestration code in SAFeSWL that defines a protocol of activation for the action names on their ports, which must be followed by their implementers. These codes are called *internal workflows*. In the certification procedure, they are combined, in parallel, with the main orchestration code of the *workflow* component. The internal workflow of Reducer is:

```
0 <parallel>
1 <iterate port="task_reduce" loop="READ_CHUNK" until="FINISH_CHUNK">
2 <invoke port= "task_reduce" action="PERFORM"/>
3 </iterate>
4 <iterate>
5 <invoke port= "task_reduce" action="CHUNK_READY"/>
6 </iterate>
7 </parallel>
```

The internal workflow of Mapper is similar, just replacing "`task_reduce`" with "`task_map`". Mappers and reducers receive chunks of input pairs in the first iteration (**read_chunk**/**finish_chunk** activation) and process them (invocation to the mapping or reduction function) when the action **perform** is activated. In parallel, as the output pairs are produced, they are sent through the **feed_pairs** port. When a sufficient number of output pairs to form a chunk is reached, the current output chunk of pairs is finished by sending an *end of chunk* marker. **chunk_ready** is activated whenever a new output chunk is started.

Shuffler (similarly, Splitter) has the following internal workflow:

```
0 <parallel>
1 <iterate>
2 <sequence>
3 <invoke port="task_shuffle_collector_active_status" action="BEGIN_CHANGE_STATUS"/>
4 <invoke port="task_shuffle_collector_active_status" action="ACTIVE|INACTIVE"/>
5 <invoke port="task_shuffle_collector_active_status" action="END_CHANGE_STATUS"/>
6 </sequence>
7 </iterate>
8 <iterate>
9 <iterate port="task_shuffle_collector_read_chunk" loop="READ_CHUNK" until="FINISH_CHUNK">
10 <invoke port="task_shuffle_feeder_read_chunk" action="READ_CHUNK"/>
11 </iterate>
12 <invoke port="task_shuffle_feeder_read_chunk" action="FINISH_CHUNK"/>
13 </iterate>
14 <iterate>
15 <invoke port="task_shuffle_feeder_chunk_ready" action="CHUNK_READY"/>
16 </iterate>
17 </parallel>
```

The only difference, compared to mappers and reducers, is the activation of the action names that enable or disable collector facets.

6.2.4. MapReduce ad hoc properties

The MapReduce *ad hoc* properties employed in this case study compose an illustrative, reduced set of properties, divided into a *safety* and a *liveness* group. The former describes precedences of execution between two distinct components or component actions. Two examples for the non-iterative workflow, among a list of six, are commented below for illustrative purposes:

$$\text{S1}: \; [!compute(23,0) * .compute(24,0)]\text{false}$$
$$\text{S2}: \; [\text{true} * .compute(24,0).!compute(24,1) * .compute(24,0)]\text{false}$$

The numbers in the formulas map to components and action name identifiers in the SAFeSWL code. The property S1 states that the action **read_chunk** (0) of binding **task_reduce_0** (24) must be preceded by the action **chunk_ready** (0) of **task_shuffle_0_feeder_chunk_ready** (23). In turn, the property S2 expresses that the action **perform** (1) of **task_reduce_0** must be activated between two executions of **read_chunk** in that component.

The liveness group includes a broader set of properties. Three examples of properties are:

$$\text{LIV1}: \; \langle true * .guard(21,0)\rangle \, \text{true}$$
$$\text{LIV2}: \; \forall c : Nat, a : Nat.\, \nu X.\mu Y.[compute(c,a)]Y \,\&\&\, [!compute(c,a)]X$$
$$\text{LIV3}: \; [\text{true}*](\forall c : Nat, a : Nat => \mu Y.([!compute(c,a)]Y \,\&\& < \text{true} > \text{true}))$$

LIV1 ensures the existence of workflow traces including the activation of the action **finish_chunk** of binding **task_shuffle_0_collector_read_chunk**. In turn, LIV2 expresses the fact that an action can only be executed along non-consecutive periods. Finally, LIV3 ensures no starvation, 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 components and actions.

A total of 18 formal properties, among default and *ad hoc* ones, have been distributed among the units of the tactical component to be proven, for both workflows. Due to the large number of components and their parallel activations in these workflows, coupled with the explosion of states generated by the model checking technique, a high verification time was

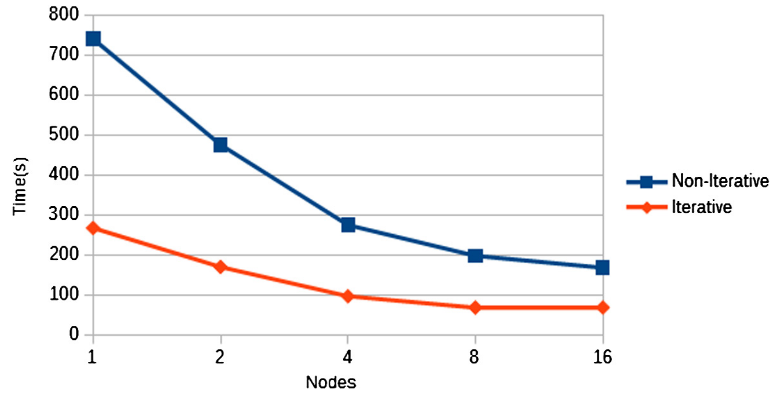

Fig. 18. Certification times for the non-iterative and iterative workflows.

Table 5
Speedups due to parallel certification.

Nodes	Non-iterative	Iterative
2	1.5	1.6
4	2.7	2.8
8	3.7	3.9
16	4.4	3.9

expected for these certifications. For example, the verification of deadlock absence and LIV3 took approximately 90 and 70 seconds each, respectively, in the non-iterative workflow.

Fig. 18 depicts the average certification times for both workflows by varying the number of units (processing nodes) of the tactical component from 1 to 16. In turn, Table 5 presents the corresponding parallel speedups. The certification times have been significantly reduced by increasing the number of units of the tactical component, despite the speedup has been sub-linear. This is explained by the fact that the certification time is limited in each scenario by the sum of the verification times of properties manipulated sequentially by the tactical component.

6.3. Parallel sorting

Parallel sorting is often used in HPC systems when dealing with huge amounts of data [49]. Thus, parallel sorting algorithms are natural candidates to be provided by developers of computation components, in HPC Shelf.

Due to the potential heterogeneity of modern distributed-memory parallel computing platforms, varying in network topologies, memory hierarchy and parallelism levels, possibly combining support for multiprocessors, multicore processors and computational accelerators (e.g. GPUs [50], MICs [51], FPGAs [52], etc.), different parallel sorting algorithms may be applied according to the execution context, as well as according to the characteristics of the data per se, such as local data pre-ordering or expected interval. Thus, the importance of the contextual contract system of HPC Shelf to support this necessary contextual abstraction becomes clear.

Consider the following contextual signature of an abstract component called Sorting, representing a family of concrete components, each one representing a particular parallel implementation of a well-known sorting algorithm, such as Quicksort, Mergesort, Bitonic Sort, Heapsort, Radix Sort, and so on [49,53,54]:

$$
\textsc{Sorting}\left[\begin{array}{l}
\textbf{\textit{sorting_place}} = S : \textsc{SPType}, \\
\textbf{\textit{pre_ordering}} = P : \textsc{POType}, \\
\textbf{\textit{data_interval}} = D : \textsc{DIType}, \\
\textbf{\textit{sorting_strategy}} = C : \textsc{CSType}\ [\textbf{\textit{pre_ordering}} = P, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{\textit{data_interval}} = D], \\
\textbf{\textit{number_nodes}} = N : \textsc{Integer}, \\
\textbf{\textit{muticore}} = M : \textsc{MCType}, \\
\textbf{\textit{accelerator_type}} = A : \textsc{AcceleratorType}\ [\textbf{\textit{muticore}} = M], \\
\textbf{\textit{topology_type}} = T : \textsc{TopologyType}
\end{array}\right]
$$

The contextual signature of Sorting declares a set of context parameters that may guide the choice of a sorting component that implement the supposedly better algorithm according to the contextual contract. They are described below:

- ***sorting_place*** states whether internal or external sorting must be employed. In *internal sorting*, the items to be sorted are placed and sorted entirely in the main memory. In turn, in *external sorting*, they are stored outside the main memory (e.g. hard disk) and loaded in small chunks;
- ***pre_ordering*** indicates the kind of local pre-ordering (e.g. a bitonic sequence) in which the items are partially sorted, if one exists;
- ***data_interval*** indicates the range of item keys (e.g. an interval of integers);
- ***sorting_strategy*** indicates which sorting strategy is employed by the component. It can be based on comparisons, where it sorts a list by repeatedly comparing pairs of keys. Contrariwise, it may employ a noncomparison-based algorithm. In such a case, for example, it may use some special *a-priori* known properties of the keys (e.g. the keys are integers from a fixed interval). This parameter is associated with parameters ***pre_ordering*** and ***data_interval***, respectively through the context variables P and D, such that the chosen sorting strategy must take into account the assumed pre-ordering and interval;
- ***number_nodes*** contains the number of processing nodes (processors) that the component expects to use during computation;
- ***muticore*** says whether the component has multi-threaded units that may employ the multiple cores of each processing node, if it is equipped with a multi-core processor or multiprocessor;
- ***accelerator_type*** represents the family of computational accelerators (GPUs, MICs, FPGAs, etc.) for which the component may extract computational power. It is associated with the parameter *muticore*, by the context variable M, so that if the processing nodes support multicore executions, the accelerators should also support the launch of multiple kernels (employing GPU terminology). The contrary is also true;
- ***topology_type*** determines the interconnection topology of the parallel computing platform hosting the component.

In the terminology of the contextual contract system, the context parameters ***sorting_strategy***, ***pre_ordering***, ***data_interval***, and ***sorting_strategy*** are *application parameters*, since they describe requirements of the system with respect to the component implementation. In turn, ***number_nodes***, ***muticore***, and ***accelerator_type*** are so-called *platform parameters*, since they describe properties of the underlying parallel computing platform that must be taken into account in the component implementation.

6.3.1. Certifying parallel sorting components

Let QuickSortImpl and MergeSortImpl be two concrete components of SORTING that implement parallel versions of the well-known *Quicksort* and *Mergesort* algorithms, respectively. They have similar contextual contracts. Indeed, the contextual contract of QuickSortImpl is:

$$\textsc{Sorting}\left[\begin{array}{l} \textbf{\textit{sorting_place}} = \textsc{InternalDistributedMemory}, \\ \textbf{\textit{pre_ordering}} = \textsc{NoPreOrdering}, \\ \textbf{\textit{data_interval}} = \textsc{NoDataInterval}, \\ \textbf{\textit{sortting_strategy}} = \\ \quad \textsc{ComparisonBased}\ [\textbf{\textit{pre_ordering}} = \textsc{NoPreOrdering}, \\ \qquad\qquad\qquad\qquad \textbf{\textit{data_interval}} = \textsc{NoDataInterval}], \\ \textbf{\textit{number_nodes}} = 4,\ \textbf{\textit{muticore}} = \textsc{NoMultipleCores}, \\ \textbf{\textit{accelerator_type}} = \textsc{NoAccelerator}\ [\textbf{\textit{muticore}} = \textsc{SingleCore}], \\ \textbf{\textit{topology_type}} = \textsc{NoTopology} \end{array}\right]$$

As one can see in the contract of QuickSortImpl (and MergeSortImpl), it performs internal sorting through distributed-memory, does not require any pre-ordering, does not make interval assumptions, employs a comparison-based sorting strategy, requires at least 4 processing nodes where it will launch its units, does not take advantage of multiple cores, does not use computational accelerators and does not assume particularities of any specific interconnection topology.

Both components employ the MPI library for enabling parallelism. Also, they employ a pure divide-and-conquer strategy, in which unit 0 has initially the vector to be sorted and then distributes the vector elements among the other units (through the `MPI_Scatter` operation). They sort locally their respective slices (through sequential Quicksort or Mergesort algorithms, respectively). At the end of this process, unit 0 receives back the sorted parts from all units (through the `MPI_Gather` operation) and performs the intercalation of the values, obtaining the final sorted vector.

For the certification of QuickSortImpl and MergeSortImpl, a parallel computing system has been created exclusively to certify them by means of the C4MPICOMPLEX certifier. Fig. 19 shows its certification architecture. Virtual platforms containing 20 processing nodes have been chosen for both components during our experiments (relevant here only to reduce search space in model checking). Also, virtual platforms containing 2 processing nodes have been chosen for all tactical components.

At the end of the certification process, the certifier has accounted for the result of the certification of the parallel sorting components. For both QuickSortImpl and MergeSortImpl, all default properties of C4MPICOMPLEX have been proved successfully. Indeed, it has been proved that they do not present deadlocks, object leaks, communication races, irrelevant barriers, out-of-bound arrays, and memory leaks and improper pointer dereference of arithmetic. Also, three component properties, annotated in the programs, have been proved:

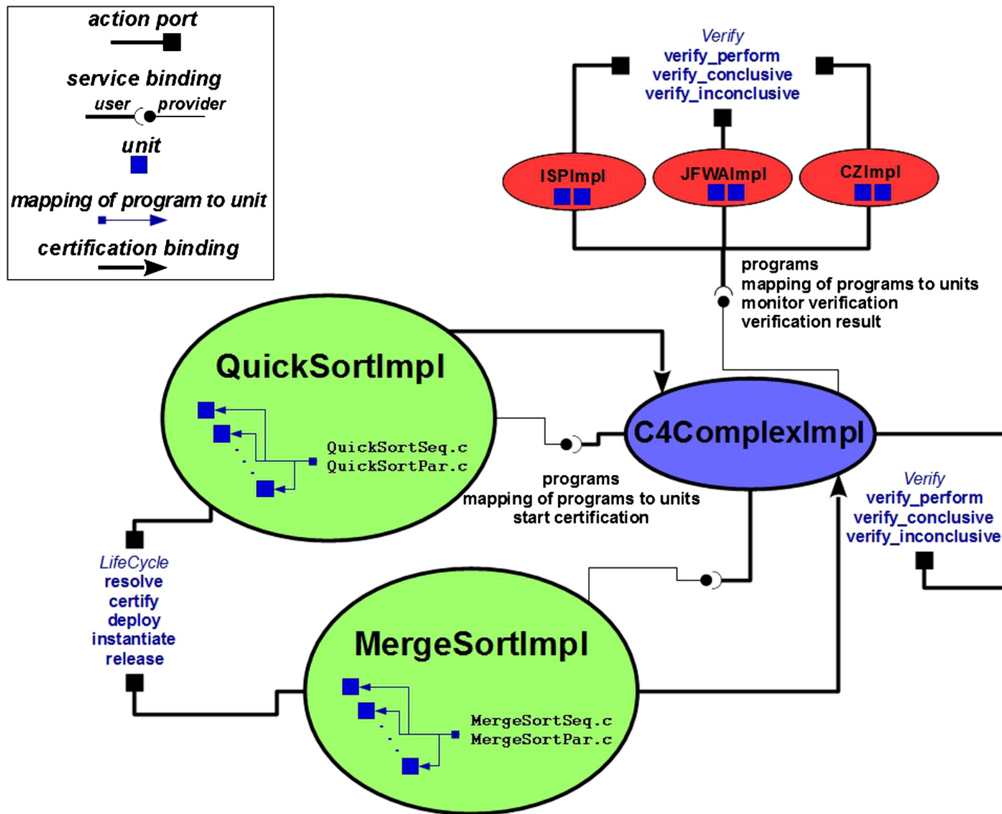

Fig. 19. Certification architecture of QuickSortImpl and MergeSortImpl.

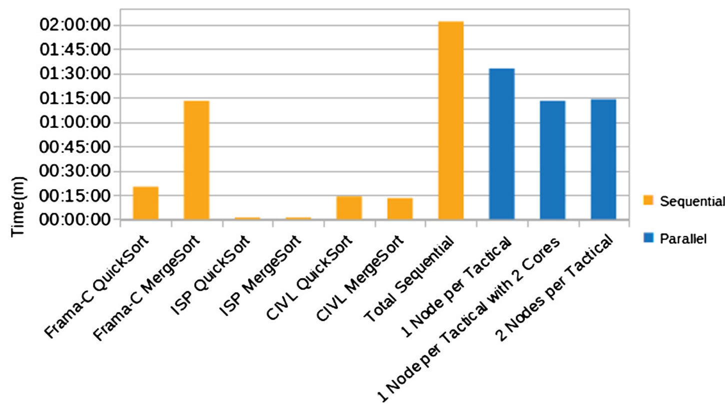

Fig. 20. Execution times for the certification of sorting components (in minutes).

- `assert forall (int i : 0..array_size-2) final_array[i]<=final_array[i+1];`
- `sorted(A,le,ri+1);`
- `permut{Old,Here}(A,le,ri).`

Fig. 20 denotes the experimental times obtained for the certification task described in this section. The individual verification times of each verification tool (direct call) and the total sequential time are shown in yellow. Parallel times, corresponding to one node (unit) per tactical component, one node per tactical component using two cores (threads) and two nodes per tactical component, are represented in blue. As it can be seen, the deductive verification performed by Frama-C for MergeSort was very costly in relation to the others. Because each verification call is indivisible in principle, the parallel times have been limited by that time. Anyway, the parallel certification was very close to that time. Finally, a greater degree of parallelism could be achieved if a verification tool could prove verification conditions in parallel. As far as we know, this is possible only through the Z3 prover, although this project has been discontinued a few years ago.

The parallel times calculated for this case study makes it possible to conclude that, in general, the smallest times happen for tactical components with a single unit running in a processing node with many cores. This is due to the fact that, in most tactical components, the verification tasks were lightweight and could be performed faster in cores with high clock frequencies.

6.4. Discussion

The case studies with Montage, MapReduce and Integer Sorting are primarily aimed at demonstrating the feasibility of certifying components of distinct kinds using the certification framework of HPC Shelf. This is the reason why we have proposed C4 and SWC2 as proof-of-concept certifiers, targeting components kinds of very different natures, i.e. computations and workflows. Once all the certification processes involved in the case studies have been completed, the experiment has been successful to demonstrate this. Therefore, it is important to emphasize that the experiments whose results are evaluated in this article do not have the ambition to constitute a definitive validation study of the certification framework of HPC Shelf.

The case studies have also shown how the inherent parallelism supported by the certification framework, using the parallel computing infrastructure of HPC Shelf itself, may be used to accelerate certification tasks, even if the underlying certification tools have not been developed with parallelism in mind, which is the case of the theorem provers and model checkers used in the experiments. Indeed, parallelism may be even more valuable in real scenarios, where more source-code and more complex orchestrations of certification tools can lead to much higher certification times. To reinforce this expectation, it is worth noting that, despite the current implementation of the certification framework is not optimal in relation to performance, the certification times achieved in the experiments, varying between 20 seconds and 12 minutes, are not influenced by possible implementation overheads. In fact, most of the certification time is spent by the underlying verification tools encapsulated in the tactical components.

7. Related work

The certification framework of HPC Shelf has not been designed as an incremental evolution of some pre-existing certification framework that could have been taken as a basis. It has been developed from scratch, under its own assumptions, to meet the particular requirements of HPC Shelf. Thus, with regard to the comparison of the certification framework herein proposed with other related works, our main challenge has been to study the literature to find such related works, and then to study their characteristics in order to determine what are the new contributions and distinguishing features of the certification framework of HPC Shelf when faced with the state-of-the-art.

7.1. Certification of software components

The certification of software components is an active research area in component-based software engineering (CBSE) since the 1990's [3–5,7]. However, as pointed out by Alvaro *et al.* [6], a consensual definition for certification does not exist within the CBSE community. From the current literature, we define the certification of software components as the study and application of methods and techniques intended to provide a well-defined level of confidence that the components of a system meet a given set of requirements. Requirements may be related to different aspects of software design, such as performance, correctness, safety, security, and so on.

The proof of concept of the HPC Shelf certification framework proposed in this paper, presented in Section 6, is carried out within the context of the formal verification of software correctness. However, it is worth noting that it can be applied to the certification of a wider range of requirements that can be certified by static means, that is, by analyzing the artifacts involved in coding, compiling and deploying components. Therefore, it is not intended to certify requirements whose certification depends on the analysis of the (dynamic) execution behavior of components. For this purpose, the certification framework supports the main elements of automated certification processes described in the literature, such as: support for a notion of *certification authority* as a stakeholder in HPC Shelf; the possibility of integrating preexisting certification tools that can look at the source code of the components, including configuration and instrumentation codes possibly required by the certification authority selected by the component developer; the flexibility offered to application providers (component users) to control the level of certification and the choice of certification authorities they trust.

The literature does not mention other proposals of general-purpose certification artifacts in the context of CBHPC[13] research, which could be directly compared to the certification framework of HPC Shelf. However, it is clear the relevance of certification in the context of long-running computations managed by SWfMSs, in computational sciences and engineering applications [55], where CBHPC platforms find their applicability. In such applications, incorrect results and execution failures may cause unsustainable increases in project costs and schedules. So, costs due to bad choices in the selection of components cannot be neglected. The impact of such costs increases as more resources of cloud computing platforms are necessary, as in scenarios where IaaS providers are used, typical for HPC Shelf applications.

[13] Component-Based High Performance Computing.

Table 6
Search results in each database.

	IEEE	Scopus	ACM	Science Direct	**Total 1**	**Total 2**
1st search	9	14	5	1	19	4
2nd search	64	79	51	10	151	15

7.2. *Verification-as-a-service (VaaS)*

As pointed out earlier, the kind of certification focused on this paper is the verification of functional and behavioral properties of components of parallel computing systems in a cloud environment through formal methods, automated by deductive and model-checking tools. Therefore, the certification framework can be evaluated in the context of VaaS [56]. In fact, it is the first proposal of VaaS framework geared to HPC requirements.

The concept of VaaS has been firstly introduced by Schaefer and Sauer to deal with scale issues of software formal verification in large-scale computational systems [56]. It is based on the notion of verification workflows, which can be executed on service-oriented computing environments, aiming at reducing the complexity of verification services. Starting from a system and properties to be verified, a verification workflow can be executed through the call of verification tasks in an appropriate order. In such a context, cloud computing is viewed as the most viable alternative to implement VaaS architectures, since it provides a service-oriented environment and a very powerful shared processing infrastructure through an abstract interface. Testing as a Service (TaaS), proposed by Candea, Bucur and Zamfir [57], follows similar principles, but in a wider, more general sense.

We have systematically searched for related work on VaaS in the most comprehensive databases of scientific literature in computer science: IEEE,[14] Scopus,[15] ACM[16] and Science Direct,[17] applying the search string *"(platform OR framework) AND service AND formal AND verification AND component AND cloud"* to title, abstract and keywords fields. The first search returned a total of 29 papers, whose abstracts and introductions were read to filter only works directly related to the certification framework of HPC Shelf. Most of the discarded papers do not propose platforms or frameworks for the intended purpose. Other discarded papers refer to conference proceedings rather than papers themselves. Finally, some papers describe work in the initial stage. In the end, we have found only 4 related papers, among which one of our previous publications [58].

In order to expand our base of comparison, and considering that the innovative essence of our work is related to VaaS, conceptually proposed for clouds, we have repeated the search after removing the term *"component"* from the search string. After that, 151 distinct papers were returned. After reading their abstracts and introductions, applying the same filter criteria, we have selected 15 related papers.

Table 6 summarizes the results found in each scientific database considered in the search. The column **Total 1** represents the number of distinct papers found, i.e. after removing redundancies. In turn, **Total 2** represents the number of papers after applying the filtering criteria. The 15 related papers found in the second search have been divided into two groups:

- frameworks or platforms for the verification of cloud administration aspects (non-functional requirements), such as elasticity, self-provisioning of resources, migration of virtual machines, etc.;
- frameworks or platforms for verification of functional requirements, which deserve special attention because they are closer to the kind of certified properties in the case studies presented in this article.

The following sections (7.2.1 and 7.2.2) describe the papers classified in these two groups, respectively. They include also some known related papers that have not been reached by the systematic search.

7.2.1. *Verification of cloud administration concerns*

Evangelidis *et al.* propose a probabilistic verification scheme aimed at dynamically evaluating auto-scaling policies of IaaS and PaaS virtual machines in Amazon EC2 and Microsoft Azure [59]. For that, it applies a Markov model implemented in the PRISM model checker [60].

Zhou *et al.* propose a formal framework for resource provisioning as a service (RPaaS) [61]. The RPaaS framework includes three modules: client, service manager and resource service. Their actions are consistent with respect to properties describing each service scenario. The UPPAAL model checker is used for verification.

Al-Haj and Al-Shaer propose a formal framework so-called VMM-Planner, for planning migrations of virtual machines (VM) [62]. Basically, it encodes a migration planning problem as a Constraint Satisfaction Problem (CSP). The initial VM placement, the target placement and a set of migration safety conditions are modeled as boolean constraints, submitted to SMT solvers.

[14] https://ieeexplore.ieee.org.
[15] https://www.scopus.com.
[16] https://dl.acm.org.
[17] https://www.sciencedirect.com.

Di Cosmo *et al.* propose the Aeolus component model [63]. It is specifically designed to capture realistic scenarios derived from configuration and deployment of applications in cloud environments. Its formal component model may describe and inspect component characteristics such as dependencies and conflicts, as well as non-functional requirements, such as reconfiguration, replication requests and load limits.

Brogi, Canciani and Soldani propose *fault-aware management protocols* for modeling the management of the behavior of application components. It attempts to analyze and automate the overall management of a multi-component application [64]. Barrel is a proof-of-concept application for integrating fault-aware management protocols in TOSCA[18] (Topology and Orchestration Specification for Cloud Applications), a OASIS standard [65].

Kumar *et al.* describe their experience of enabling the Microsoft Static Driver Verifier (SDV)[19] to use the Microsoft Azure cloud computing platform [66]. The architecture and methodology for enabling SDV to operate in Azure, as well as the results of SDV on single drivers and driver suites using various configurations of the cloud relative to a local machine are reported.

Finally, Sahli *et al.* propose a semantic framework based on bigraphical reactive systems (BRS) [67] and Maude language [68] for modeling both structural and behavioral aspects of cloud-based systems, aimed at verifying elasticity properties inherent to these systems through model checking [69].

7.2.2. Verification of functional requirements

Nezhad *et al.* propose COOL, a framework for provider-side design of cloud solutions based on formal methods and model-driven engineering [70]. The approach makes it easier the job of automating the generation of solutions, from client requirements to a complete and correct cloud solution.

Moscato *et al.* propose a Model Driven Engineering (MDE) approach in the context of the MetaMORP(h)OSY framework [71], intended to automatic generation of monitors of formal properties to be verified during the entire life-cycle of cloud components [72].

Chen *et al.* propose a formal verification framework that automatically detects conflicts concerning enterprise policies and inconsistencies in user requirements, specified through constraint programming and checked through SMT solvers [73]. A system of automatic cloud services selection chooses those ones satisfying all enterprise policies and user requirements.

Klai and Ochi address the problem of abstracting and verifying the correctness of integrating service-based business processes (SBPs) [74]. The formal system employs a bottom-up approach in order to check the consistency of several kinds of interactions, including composition, asynchronous communications and resource sharing. Properties are expressed in Linear Temporal Logic (LTL) [75] and verified through model checking.

Akhunzada *et al.* produced a formal framework for a service broker [76], helping to compose formally described QoS metrics by following the workflow-based nature of web services composition. Functional and non-functional requirements derived from the composition process are specified using a variant of π-calculus [77].

Montesi and Sangiorgi introduce a model of components following the process calculus approach [78]. The work consisted of isolating primitives that capture the relevant concepts of component-based systems, including: a hierarchical structure of components; a prominent role to input/output interfaces; the possibility of stopping and capturing components; a mechanism of channel interactions, orthogonal to the activity of components.

The approach proposed by Beyer *et al.* evaluates Google App Engine as a verification infrastructure, by means of porting the open-source verification framework CPAchecker to it [79]. A new verification service was proposed as a web front-end to users who wish to perform single verification tasks, and an API for integrating the service into existing verification infrastructures.

The work of Chunling Hu *et al.* proposes a method for verifying cloud application responses (actions) to user requests [80]. Applications are modeled in SoaML (Service-Oriented Architecture Modeling Language), service interfaces are translated into PROMELA [81], and service contracts are described in terms of LTL formulas [75]. Both PROMELA programs and LTL formulas are integrated into the SPIN [81] model checker for verification.

Skowyra *et al.* present Verificare, a verification platform for applications based on Software-Defined Networks (SDN) [82]. SDN components, safety and security requirements, can be specified from a variety of formal libraries and automatically translated and verified through a variety of tools, such as PRISM [60], SPIN [81] and Alloy [83].

Ren *et al.* propose a formal approach for modeling and verifying distributed systems that integrates UML sequence diagrams, π-calculus [77] and the symbolic model checker NuSMV [84,85]. Three layers compose the framework: *graphical layer*, which uses sequence diagrams for system modeling; *formal specification layer*, which uses π-calculus to formalize the UML sequence diagram; and *verification layer*, in which π-calculus processes are verified by NuSMV.

Ciortea *et al.* propose Cloud9, a cloud-based testing service that promises to make high-quality testing fast, cheap, and practical [86]. Despite being reported as a testing web service, it parallelizes symbolic execution, a popular model checking technique, to run on large shared-nothing clusters of computers, such as Amazon EC2.

Mancine *et al.* propose a verification service to show system correctness in regard to uncontrollable events, through an exhaustive hardware simulation by taking into account all relevant scenarios [87].

[18] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[19] http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808.aspx.

Belletine *et al.* propose a distributed framework for verifying CTL formulas on a cloud, based on a MapReduce algorithm [88].

Finally, the framework proposed by Kai Hu *et al.* propose a robust VaaS framework, focusing essentially on the dualism with the main concerns of SaaS (Software-as-a-Service), such as the storage of verification tools and results, scalability problems and fault tolerance [89]. It is the closest framework to the certification framework of HPC Shelf. However, it assumes the availability of verification tools in the cloud in a raw way, requiring, for application developers, experience with the use of the tool, which is, in general, an unrealistic assumption.

7.3. Discussion

In comparison with the related work described above, the certification framework of HPC Shelf has the following distinguishing characteristics:

- It is a general-purpose framework that can be used for automatic certification of a wide range of requirements, including both functional and non-functional, while other works address a particular requirement.
- It allows the integration of an arbitrary set of different automatic verification tools necessary for certification, through component encapsulation. In other works, it is supported a fixed set of tools that are appropriate for the requirements to be certified.
- It makes use of HPC techniques to accelerate certification tasks, as well as to address bigger verification problems, such as in the case of state explosion problems in model-checking [90]. For that, it may employ the same parallel computing infrastructure where certifiable components perform their tasks. It is possible to exploit different levels of parallelism supported by virtual platforms (distributed-memory, shared-memory, multi/many-core, accelerators, etc.) even when the underlying verification tools encapsulated in tactical components are sequential.
- It supports a seamless separation of concerns among stakeholders involved in the certification process, as explained in the next paragraph.

Contrariwise to the framework of Kai Hu *et al.*, the certification framework of HPC Shelf promotes a seamless separation of concerns among the stakeholder classes involved in certification: application providers, component developers, and certification authorities. The expertise in formal methods and verification tools is an intrinsic characteristic of certification authorities. The application provider has only the responsibility of setting *adhoc* and contractual properties for the certifier prescribed by the developer of the component it will use, in the format prescribed by the certifier's documentation. In turn, certifier selection is a component developer responsibility, using contextual contracts. Although such a selection does not determine which certification authority will actually provide the implementation of the certifier used in the execution, it is still possible to delegate responsibilities to guide the selection of certification authorities to the application provider itself, partially or totally, through the contextual contract of the certifier. Also, when designing certifier components, certification authorities may provide high-level interfaces to facilitate the interaction of application providers and component developers with the underlying verification tools.

7.4. Comparison with prior proceedings papers

The work with the certification framework of HPC Shelf has been presented in two conferences, namely CLOSER'2017 [58] and FACS'2017 [91]. The papers published in their proceedings are most focused on the design of C4 and SWC2 certifiers, respectively. Finally, this paper consolidates the results achieved by this work, by refining and expanding the results presented in these papers. In particular, it introduces the pair of C4 certifiers so-called C4MPISimple and C4MPIComplex to demonstrate, using the case studies Montage and Sorting, how different C4 certifiers may be developed with distinct purposes. Also, the MapReduce case study has been completely reformulated, by using to the most recent version of SAFeSWL and a more complex workflow compared to the one presented to the FACS'2017 audience. Finally, there are some modifications in the framework architecture, such as the introduction of services ports for the communication between the certifier and tactical components.

8. Conclusions and future work

This article has proposed a certification framework for HPC Shelf, aimed at certifying components of different kinds with respect to given set of requirements. By using it, different sorts of certification components, with interfaces to existing formal verification tools, can be added to a parallel computing system. Certification of isolated components, including workflows that orchestrate parallel computing systems, is therefore provided as another service in the cloud, in a truly reflexive way. In this sense, a *parallel certification system* is built according to the same architectural and operational principles governing the design of parallel computing systems of HPC Shelf. Thus, certifier and certifiable components can be orchestrated in a similar way, and their executions can overlap without interference.

The certification framework discussed in the article is an ongoing project whose initial, proof-of-concept prototype was developed in C#/MPI and validated through the case studies presented in Section 6.

By its architectural principles, the framework scales easily to bigger and more demanding examples. As a matter of fact, one of its main characteristics is *extensibility*, since new certifier and tactical components can be added on-demand, in order to deal with the certification of different component kinds and requirements. For instance, there is a plan to design certifiers for connector components by using the Reo component model [92]. Also, there are plans to work with certification of certain non-functional dependability requirements [93], such as quality of service (QoS) and fault tolerance. However, contrariwise to the case studies with SWC2 and C4, whose only objectives have been to perform proof-of-concept validation on the proposed certification framework, it is intended to develop state-of-the-art certifiers.

From a broader perspective, the inclusion, at the design time of an application, of components that in a reflexive way are able to inspect and certify other components proved worth to explore. First the inherent complexity of cloud-based applications, given the heterogeneity of resources and their open and poly-centric control entails the need for scaling up formal verification tools. On the other hand, the cloud itself is an ecosystem in which components that orchestrate verification engines may live and be invoked to certify their own computations and the ways they interact. While the former perspective has already been the focus of a number of research initiatives (see, for example, the ABS project [94]), the second one constitutes an open challenge to the software engineering community.

Appendix A. XSD grammar of SAFeSWL

```
0  <schema>
1  <element name="workflow" type="tns:SAFeSWL_OperationAnyType"/>
2
3  <complexType name="SAFeSWL_OperationManyType">
4  <complexContent>
5  <extension base="tns:SAFeSWL_OperationBaseType">
6  <choice maxOccurs="unbounded" minOccurs="1">
7  <element name="skip"     type="tns:SAFeSWL_OperationPrimitiveType"/>
8  <element name="break"    type="tns:SAFeSWL_OperationPrimitiveType"/>
9  <element name="continue" type="tns:SAFeSWL_OperationPrimitiveType"/>
10 <element name="start"    type="tns:SAFeSWL_OperationPrimitiveInvokeActionAsyncType"/>
11 <element name="wait"     type="tns:SAFeSWL_OperationPrimitiveInvokeActionAsyncType"/>
12 <element name="cancel"   type="tns:SAFeSWL_OperationPrimitiveInvokeActionAsyncType"/>
13 <element name="invoke"   type="tns:SAFeSWL_OperationPrimitiveInvokeActionType"/>
14 <element name="sequence" type="tns:SAFeSWL_OperationManyType"/>
15 <element name="parallel" type="tns:SAFeSWL_OperationManyType"/>
16 <element name="choice"   type="tns:SAFeSWL_OperationChoiceType"/>
17 <element name="iterate"  type="tns:SAFeSWL_IterateType"/>
18 </choice>
19 </extension>
20 </complexContent>
21 </complexType>
22
23 <complexType name="SAFeSWL_OperationBaseType">
24 <attribute name="order" type="int" use="optional"/>
25 <attribute name="value" type="string" use="optional"/>
26 <attribute name="oper_name" type="string" use="optional"/>
27 <attribute name="level" type="int" use="optional"/>
28 <attribute name="base_label" type="string" use="optional"/>
29 <attribute name="tracing" type="boolean" use="optional" default="false"/>
30 </complexType>
31
32 <complexType
33 name="SAFeSWL_OperationPrimitiveInvokeActionType">
34 <complexContent>
35 <extension
36 base="tns:SAFeSWL_OperationPrimitiveType">
37 <attribute name="port" type="string"/>
38 <attribute name="action" type="string" use="required"/>
39 </extension>
40 </complexContent>
41 </complexType>
42
43 <complexType
44 name="SAFeSWL_OperationPrimitiveInvokeActionAsyncType">
45 <complexContent>
46 <extension
47 base="tns:SAFeSWL_OperationPrimitiveInvokeActionType">
48 <attribute name="handle_id" type="string" use="optional"/>
49 </extension>
50 </complexContent>
51 </complexType>
52
53 <complexType name="SAFeSWL_OperationAnyType">
54 <complexContent>
55 <extension base="tns:SAFeSWL_OperationBaseType">
56 <choice maxOccurs="1" minOccurs="0" >
57 <element name="skip" type="tns:SAFeSWL_OperationPrimitiveType"/>
58 <element name="break" type="tns:SAFeSWL_OperationPrimitiveType"/>
59 <element name="continue" type="tns:SAFeSWL_OperationPrimitiveType"/>
60 <element name="start" type="tns:SAFeSWL_OperationPrimitiveInvokeActionAsyncType"/>
61 <element name="wait" type="tns:SAFeSWL_OperationPrimitiveInvokeActionAsyncType"/>
62 <element name="cancel" type="tns:SAFeSWL_OperationPrimitiveInvokeActionAsyncType"/>
63 <element name="invoke" type="tns:SAFeSWL_OperationPrimitiveInvokeActionType"/>
64 <element name="sequence" type="tns:SAFeSWL_OperationManyType"/>
65 <element name="parallel" type="tns:SAFeSWL_OperationManyType"/>
66 <element name="choice" type="tns:SAFeSWL_OperationChoiceType"/>
67 <element name="iterate" type="tns:SAFeSWL_IterateType"/>
68 </choice>
69 </extension>
70 </complexContent>
71 </complexType>
```

```
72
73 <complexType name="SAFeSWL_OperationChoiceType">
74 <complexContent>
75 <extension base="tns:SAFeSWL_OperationBaseType">
76 <sequence>
77 <element name="select" type="tns:SAFeSWL_SelectionGuardType" maxOccurs="unbounded" minOccurs="1"/>
78 </sequence>
79 <attribute name="port" type="string"/>
80 </extension>
81 </complexContent>
82 </complexType>
83
84 <complexType name="SAFeSWL_SelectionGuardType">
85 <complexContent>
86 <extension base="tns:SAFeSWL_OperationAnyType">
87 <attribute name="action" type="string"/>
88 </extension>
89 </complexContent>
90 </complexType>
91
92 <complexType name="SAFeSWL_OperationPrimitiveType">
93 <complexContent>
94 <extension base="tns:SAFeSWL_OperationBaseType"/>
95 </complexContent>
96 </complexType>
97
98 <complexType name="SAFeSWL_ConditionType">
99 <sequence>
100 <element name="condition" type="tns:SAFeSWL_OperationPrimitiveInvokeActionType" maxOccurs="unbounded"
        minOccurs="1"/>
101 </sequence>
102 </complexType>
103
104 <complexType name="SAFeSWL_IterateType">
105 <complexContent>
106 <extension base="tns:SAFeSWL_OperationAnyType">
107 <choice maxOccurs="1" minOccurs="0">
108 <element name="branch" type="tns:SAFeSWL_BranchType" maxOccurs="unbounded" minOccurs="1"/>
109 <element name="select" type="tns:SAFeSWL_SelectionGuardType" maxOccurs="unbounded" minOccurs="1"/>
110 </choice>
111 <attribute name="port" type="string" use="optional"/>
112 <attribute name="loop" type="string" use="optional"/>
113 <attribute name="until" type="string" use="optional"/>
114 </extension>
115 </complexContent>
116 </complexType>
117
118 <complexType name="SAFeSWL_BranchType">
119 <complexContent>
120 <extension base="tns:SAFeSWL_OperationAnyType">
121 <sequence>
122 <element name="select" type="tns:SAFeSWL_SelectionGuardType" maxOccurs="unbounded" minOccurs="0"/>
123 </sequence>
124 <attribute name="port" type="string"/>
125 <attribute name="loop" type="string" use="optional"/>
126 <attribute name="until" type="string"/>
127 </extension>
128 </complexContent>
129 </complexType>
130 </schema>
```

References

[1] J.C. Silva, A.B.O. Dantas, F.H. de Carvalho Junior, A Scientific Workflow Management System for orchestration of parallel components in a cloud of large-scale parallel processing services, Sci. Comput. Program. 173 (15) (2019) 1058–1074.

[2] P. Calegari, M. Levrier, P. Balczyński, Web portals for high-performance computing: a survey, ACM Trans. Web 13 (1) (2019) 5:1–5:36, https://doi.org/10.1145/3197385.

[3] C. Wohlin, P. Runeson, Certification of software components, IEEE Trans. Softw. Eng. 20 (6) (1994) 494–499, https://doi.org/10.1109/32.295896.

[4] J.M. Voas, Certifying off-the-shelf software components, Computer 31 (6) (1998) 53–59.

[5] J. Morris, G. Lee, K. Parker, G.A. Bundell, Software component certification, Computer 34 (9) (2001) 30–36, https://doi.org/10.1109/2.947086.

[6] A. Alvaro, E.S. de Almeida, S.R.L. Meira, Software component certification: a survey, in: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005, pp. 106–113, https://doi.org/10.1109/EUROMICRO.2005.52.

[7] J. Boegh, Certifying software component attributes, IEEE Softw. 23 (3) (2006) 74–81.

[8] F.H. de Carvalho Junior, R. Lins, R.C. Correa, G.A. Araújo, Towards an architecture for component-oriented parallel programming, Concurr. Comput.: Pract. Exp. 19 (5) (2007) 697–719.

[9] F.H. de Carvalho Junior, R.D. Lins, An institutional theory for #-components, Electron. Notes Theor. Comput. Sci. 195 (2008) 113–132.

[10] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[11] C.A. Rezende, F.H. de Carvalho Junior, MapReduce with components for processing big graphs, in: 2018 Symposium on High Performance Computing Systems (WSCAD'2018), 2018, pp. 108–115, https://doi.org/10.1109/WSCAD.2018.00026.

[12] F.H. de Carvalho Junior, C.A. Rezende, J.C. Silva, W.G. Al Alam, J.M.U. de Alencar, Contextual abstraction in a type system for component-based high performance computing platforms, Sci. Comput. Program. 132 (2016) 96–128, https://doi.org/10.1016/j.scico.2016.07.005.

[13] F.H. de Carvalho Junior, C.A. Rezende, A case study on expressiveness and performance of component-oriented parallel programming, J. Parallel Distrib. Comput. 73 (5) (2013) 557–569, https://doi.org/10.1016/j.jpdc.2012.12.007.

[14] W.G. Al-Alam, F.H. de Carvalho-Junior, Contextual contracts for component-based resource abstraction in a cloud of HPC services, in: Proceedings of the XX Symposium on High Performance Computing Systems, WSCAD'2019, SBC, Porto Alegre, RS, Brasil, 2019, pp. 216–227, https://doi.org/10.5753/wscad.2019.8670.

[15] J. Dongarra, S.W. Otto, M. Snir, D. Walker, An Introduction to the MPI Standard, Tech. Rep. CS-95-274, University of Tennessee, Jan. 1995.

[16] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings, 17th Annual IEEE Symposium on Logic in Computer Science, 2002, IEEE Computer Society, 2002, pp. 55–74.

[17] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Inform. 6 (4) (1976) 319–340.
[18] K.R. Apt, Correctness proofs of distributed termination algorithms, ACM Trans. Program. Lang. Syst. 8 (3) (1986) 388–405.
[19] H.A. López, E.R.B. Marques, F. Martins, N. Ng, C. Santos, V.T. Vasconcelos, N. Yoshida, Protocol-based verification of message-passing parallel programs, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, 2015, pp. 280–298.
[20] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, VCC: a practical system for verifying concurrent C, in: Theorem Proving in Higher Order Logics, Springer, 2009, pp. 23–42.
[21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: a powerful, sound, predictable, fast verifier for C and Java, in: M. Bobaru, K. Havelund, G. Holzmann, R. Joshi (Eds.), NASA Formal Methods, Springer, Berlin, Heidelberg, 2011, pp. 41–55.
[22] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-C, in: Proceedings of the 10th International Conference on Software Engineering and Formal Methods, SEMF'2012, Springer, 2012, pp. 233–247.
[23] M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino, Boogie: a modular reusable verifier for object-oriented programs, in: Proceedings of the 4th International Conference on Formal Methods for Components and Objects, FMCO'05, Springer, 2006, pp. 364–387.
[24] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, Why3: shepherd your herd of provers, in: Boogie 2011: First International Workshop on Intermediate Verification Languages, 2011, pp. 53–64.
[25] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, A. Mebsout, The Alt-Ergo automated theorem prover, On-line, http://alt-ergo.lri.fr, 2008. (Accessed 21 June 2019).
[26] C. Barrett, C. Tinelli, CVC3, in: International Conference on Computer Aided Verification, Springer, 2007, pp. 298–302.
[27] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, CVC4, in: International Conference on Computer Aided Verification, Springer, 2011, pp. 171–177.
[28] L. De Moura, N. Bjørner, Z3: an efficient SMT solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
[29] S. Schulz, E – a brainiac theorem prover, AI Commun. 15 (2,3) (2002) 111–126.
[30] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, SPASS version 3.5, in: International Conference on Automated Deduction, Springer, 2009, pp. 140–145.
[31] A. Riazanov, A. Voronkov, Vampire, in: International Conference on Automated Deduction, Springer, 1999, pp. 292–296.
[32] The Coq development team, The Coq Proof Assistant Reference Manual, LogiCal Project, version 8.0, http://coq.inria.fr, 2004.
[33] T. Nipkow, M. Wenzel, L.C. Paulson, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Springer, Berlin, Heidelberg, 2002.
[34] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R.M. Kirby, R. Thakur, Formal verification of practical MPI programs, SIGPLAN Not. 44 (4) (2009) 261–270, https://doi.org/10.1145/1594835.1504214.
[35] S.F. Siegel, M. Zheng, Z. Luo, T.K. Zirkel, A.V. Marianiello, J.G. Edenhofner, M.B. Dwyer, M.S. Rogers, CIVL: the concurrency intermediate verification language, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 61.
[36] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, Let's verify this with Why3, Int. J. Softw. Tools Technol. Transf. (2014) 1–19, https://doi.org/10.1007/s10009-014-0314-5.
[37] J. Qin, T. Fahringer, S. Pllana, UML based grid workflow modeling under ASKALON, in: P. Kacsuk, T. Fahringer, Z. Németh (Eds.), Distributed and Parallel Systems, Springer, Boston, MA, 2007, pp. 191–200.
[38] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, J. Patel, Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling, Springer, London, 2007, pp. 428–449, Ch. 26, https://doi.org/10.1007/978-1-84628-757-2_26.
[39] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurr. Comput.: Pract. Exp. 18 (10) (2006) 1039–1065.
[40] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, et al., Pegasus: a framework for mapping complex scientific workflows onto distributed systems, Sci. Program. 13 (3) (2005) 219–237.
[41] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al., The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud, Nucleic Acids Res. 41 (W1) (2013) W557.
[42] A. Harrison, I. Taylor, I. Wang, M. Shields, WS-RF workflow in Triana, Int. J. High Perform. Comput. Appl. 22 (3) (2008) 268–283.
[43] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, M. van Weerdenburg, The formal specification language mCRL2, in: E. Brinksma, D. Harel, A. Mader, P. Stevens, R. Wieringa (Eds.), Methods for Modelling Software Systems (MMOSS), No. 06351 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2007, pp. 1–34.
[44] J.F. Groote, M.R. Mousavi, Modeling and Analysis of Communicating Systems, MIT Press, 2014.
[45] J.C.M. Baeten, T. Basten, M.A. Reniers, Process Algebra: Equational Theories of Communicating Processes, Cambridge Tracts in Theoretical Computer Science, vol. 50, Cambridge University Press, 2010.
[46] D. Sannella, A. Tarlecki, Foundations of Algebraic Specifications and Formal Program Development, Cambridge University Press, 2011.
[47] D. Kozen, Results on the propositional μ-calculus, Theor. Comput. Sci. 27 (1983) 333–354.
[48] G.B. Berriman, E. Deelman, J.C. Good, J.C. Jacob, D.S. Katz, C. Kesselman, A.C. Laity, T.A. Prince, G. Singh, M.-H. Su, Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, in: SPIE Astronomical Telescopes+ Instrumentation, International Society for Optics and Photonics, 2004, pp. 221–232.
[49] S.G. Akl, Parallel Sorting Algorithms, Academic Press, Inc., Orlando, FL, USA, 1990.
[50] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum Stover, GPU cluster for high performance computing, in: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC'04, IEEE Computer Society, 2004, pp. 47–58, https://doi.org/10.1109/SC.2004.26.
[51] A. Duran, M. Klemm, The Intel many integrated core architecture, in: Proceedings of the International Conference on High Performance Computing and Simulation, HPCS'2012, IEEE Computer Society, 2012, pp. 365–366, https://doi.org/10.1109/HPCSim.2012.6266938.
[52] M.C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, D. DiSabello, Achieving high performance with FPGA-based computing, Computer 40 (2007) 50–57, https://doi.org/10.1109/MC.2007.79.
[53] M.J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Education Group, 2003.
[54] F. Dehne, H. Zaboli, Parallel sorting for GPUs, in: Emergent Computation, Springer, 2017, pp. 293–302.
[55] I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields, Workflows for e-Science: Scientific Workflows for Grids, Springer, Secaucus, NJ, USA, 2006.
[56] I. Schaefer, T. Sauer, Towards verification as a service, in: A. Moschitti, R. Scandariato (Eds.), Eternal Systems, Springer, Berlin, Heidelberg, 2012, pp. 16–24.
[57] G. Candea, S. Bucur, C. Zamfir, Automated software testing as a service (TaaS), in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10, 2010, pp. 212–223, https://doi.org/10.1145/1807128.1807153.
[58] A.B. de Oliveira Dantas, F.H. de Carvalho Junior, 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, ScitePress, 2017, pp. 229–240.
[59] A. Evangelidis, D. Parker, R. Bahsoon, Performance modelling and verification of cloud-based auto-scaling policies, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, 2017, pp. 355–364, https://doi.org/10.1109/CCGRID.2017.39.

[60] M. Kwiatkowska, G. Norman, D. Parker, PRISM: probabilistic symbolic model checker, in: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Springer, 2002, pp. 200–204.
[61] W. Zhou, L. Liu, S. Lü, P. Zhang, Toward formal modeling and verification of resource provisioning as a service in cloud, IEEE Access 7 (2019) 26721–26730, https://doi.org/10.1109/ACCESS.2019.2900473.
[62] S. Al-Haj, E. Al-Shaer, A formal approach for virtual machine migration planning, in: Proceedings of the 9th International Conference on Network and Service Management, CNSM 2013, 2013, pp. 51–58, https://doi.org/10.1109/CNSM.2013.6727809.
[63] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Aeolus: a component model for the cloud, Inf. Comput. 239 (2014) 100–121.
[64] A. Brogi, A. Canciani, J. Soldani, Fault-aware management protocols for multi-component applications, J. Syst. Softw. 139 (2018) 189–210.
[65] A. Brogi, J. Soldani, P. Wang, Tosca in a nutshell: promises and perspectives, in: Service-Oriented and Cloud Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 171–186.
[66] R. Kumar, T. Ball, J. Lichtenberg, N. Deisinger, A. Upreti, C. Bansal, CloudSDV: enabling Static Driver Verifier using Microsoft Azure, in: IFM 2016 Proceedings of the 12th International Conference on Integrated Formal Methods, vol. 9681, Springer-Verlag New York, Inc., New York, NY, USA, 2016, pp. 523–536.
[67] R. Milner, Bigraphs and their algebra, Electron. Notes Theor. Comput. Sci. 209 (2008) 5–19, https://doi.org/10.1016/j.entcs.2008.04.002.
[68] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí Oliet, J. Meseguer, C. Talcott, All about Maude – a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Springer-Verlag, Berlin, Heidelberg, 2007.
[69] H. Sahli, F. Belala, C. Bouanaka, A BRS-based approach to model and verify cloud systems elasticity, in: 1st International Conference on Cloud Forward: From Distributed to Complete Computing, Proc. Comput. Sci. 68 (2015) 29–41, https://doi.org/10.1016/j.procs.2015.09.221.
[70] N. Motahari, T. Nakamura, A. Sosnovich, P. Yin, K. Yorav, A model-driven framework for automated generation and verification of cloud solutions from requirements, in: Service-Oriented Computing, Springer, 2018, pp. 714–721, https://doi.org/10.1007/978-3-030-03596-9_51.
[71] F. Moscato, S. Venticinque, R. Aversa, B. Di Martino, Formal modeling and verification of real-time multi-agent systems: the REMM framework, in: Intelligent Distributed Computing, Systems and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 187–196.
[72] D.D. Domenico, F. Moscato, Automatic monitor generation for cloud services, in: 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, 2015, pp. 547–552, https://doi.org/10.1109/CISIS.2015.81.
[73] C. Chen, S. Yan, G. Zhao, B.S. Lee, S. Singhal, A systematic framework enabling automatic conflict detection and explanation in cloud service selection for enterprises, in: 2012 IEEE Fifth International Conference on Cloud Computing, 2012, pp. 883–890, https://doi.org/10.1109/CLOUD.2012.95.
[74] K. Klai, H. Ochi, LTL mode checking of service-based business processes in the cloud, in: 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 3, 2015, pp. 398–403, https://doi.org/10.1109/COMPSAC.2015.251.
[75] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, Berlin, Heidelberg, 1992.
[76] A. Akhunzada, A. Gani, S. Hussain, A.A. Khan Ashrafullah, A formal framework for web service broker to compose QoS measures, in: 2015 SAI Intelligent Systems Conference, IntelliSys, 2015, pp. 532–536, https://doi.org/10.1109/IntelliSys.2015.7361191.
[77] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes, Tech. Rep. ECS-LFCS-89-85 and -86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.
[78] F. Montesi, D. Sangiorgi, A model of evolvable components, in: Trustworthy Global Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 153–171.
[79] D. Beyer, G. Dresler, P. Wedler, Software verification in the Google app-engine cloud, in: Computer Aided Verification, Springer International Publishing, Cham, 2014, pp. 327–333.
[80] C. Hu, G. Geng, B. Li, C. Tang, X. Wang, Verifying cloud application for the interaction correctness using SoaML and SPIN, in: Proceedings of the 2019 8th International Conference on Software and Computer Applications, ICSCA '19, ACM, New York, NY, USA, 2019, pp. 210–216, https://doi.org/10.1145/3316615.3316714.
[81] M. Dabaghchian, M. Abdollahi Azgomi, Model checking the observational determinism security property using PROMELA and SPIN, Form. Asp. Comput. 27 (5–6) (2015) 789–804.
[82] R. Skowyra, A. Lapets, A. Bestavros, A. Kfoury, A verification platform for SDN-enabled applications, in: Proceedings of the 2014 IEEE International Conference on Cloud Engineering, IC2E '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 337–342, https://doi.org/10.1109/IC2E.2014.72.
[83] D. Jackson, Alloy: a lightweight object modelling notation, ACM Trans. Softw. Eng. Methodol. 11 (2) (2002) 256–290.
[84] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an OpenSource tool for symbolic model checking, in: Proceedings of the 14th International Conference on Computer Aided Verification, CAV '02, Springer-Verlag, London, UK, 2002, pp. 359–364.
[85] G. Ren, P. Deng, C. Yang, J. Zhang, Q. Hua, A formal approach for modeling and verification of distributed systems, in: Cloud Computing, Springer International Publishing, Cham, 2016, pp. 317–322.
[86] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, G. Candea, Cloud9: a software testing service, Oper. Syst. Rev. 43 (2009) 5–10, https://doi.org/10.1145/1713254.1713257.
[87] T. Mancini, F. Mari, A. Massini, I. Melatti, E. Tronci, SyLVaaS: system level formal verification as a service, Fundam. Inform. 149 (2016) 101–132.
[88] C. Bellettini, M. Camilli, L. Capra, M. Monga, Distributed CTL model checking using MapReduce: theory and practice, Concurr. Comput.: Pract. Exp. 28 (11) (2016) 3025–3041.
[89] K. Hu, L. Lei, W.-T. Tsai, Multi-tenant Verification-as-a-Service (VaaS) in a cloud, Simul. Model. Pract. Theory 60 (2016) 122–143.
[90] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Progress on the state explosion problem in model checking, in: Informatics – 10 Years Back. 10 Years Ahead, Springer-Verlag, Berlin, Heidelberg, 2001, pp. 176–194.
[91] A.B. de Oliveira Dantas, F.H. de Carvalho Junior, L. Soares Barbosa, Certification of workflows in a component-based cloud of high performance computing services, in: Proceedings of the 14th International Conference on Formal Aspects of Component Software, FACS'2017, Springer, Braga, Portugal, 2017, pp. 198–215.
[92] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.
[93] J.C.C. Laprie, A. Avizienis, H. Kopetz (Eds.), Dependability: Basic Concepts and Terminology, Springer-Verlag, Berlin, Heidelberg, 1992.
[94] E. Albert, F.S. de Boer, R. Hähnle, E.B. Johnsen, R. Schlatte, S. Tarifa, P. Wong, Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS, Serv. Oriented Comput. Appl. 8 (4) (2014) 323–339.