A Framework for Certification of Large-Scale Component-Based Parallel Computing Systems in a Cloud Computing Platform for HPC Services

Allberson Bruno de Oliveira Dantas[1], Francisco Heron de Carvalho Junior[1], Luís Soares Barbosa[2]
[1]*Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Brazil*
[2]*HASLab INESC TEC & Universidade do Minho, Campus de Gualtar, Braga, Portugal*
{*allberson, heron*}*@lia.ufc.br*, {*lsb*}*@di.uminho.pt*

Keywords: Software Formal Verification; Verification as a Service (VaaS); Cloud Computing; Software Components; High Performance Computing

Abstract: This paper addresses the verification of software components in the context of their orchestration to build cloud-based scientific applications with high performance computing requirements. In such a scenario, components are often supplied by different sources and their cooperation rely on assumptions of conformity with their published behavioral interfaces. Therefore, a faulty or ill-designed component, failing to obey to the envisaged behavioral requirements, may have dramatic consequences in practice. Certifier components, introduced in this paper, implement a verification as a service framework and are able to access the implementation of other components and verify their consistency with respect to a number of functional, safety and liveness requirements relevant to a specific application or a class of them. It is shown how certifier components can be smoothly integrated in HPC Shelf, a cloud-based platform for high performance computing in which different sorts of users can design, deploy and execute scientific applications.

1 INTRODUCTION

Unlike in engineering, current software development methods usually do not provide strong assurance concerning software reliability. For this reason, great research efforts, both in academia and industry, have proposed mechanisms for formal verification of software. However, testing and *a posteriori* empirical error detection are still the usual practice.

The main reason of the poor dissemination of formal methods is the inherent complexity of computational systems. Each line of code is a potential source of errors in programs, especially parallel ones, leading to a large number of states, making it difficult to predict the behavior of an application and verify its properties. This difficulty is increased in emerging heterogeneous computing environments in High Performance Computing (HPC). In fact, the use of formal methods for improving HPC software, whose complexity has increased significantly in the recent years, is very poorly explored in research initiatives.

HPC Shelf is a proposal of cloud computing platform for providing HPC services (de Carvalho Silva and de Carvalho Junior, 2016). It offers an environment to develop applications for the needs of specialists of a given domain, dealing with domain-specific, computationally intensive problems by orchestrating parallel components whose performance are tuned to classes of parallel computing platforms. Such components form *parallel computing systems*, orchestrated through SAFe (*Shelf Application Framework*), a scientific workflow management system.

In such a scenario, computational scientists and engineers, typically with poor computer science background, need to be assured that the available components behave as declared in their published interfaces.

Therefore, certification of components, through the verification of functional and behavioral requirements relevant to applications, becomes an essential ingredient of the HPC Shelf platform, as a way to increase their confidence levels.

In this paper, we introduce a component-based *certification framework* for components of HPC Shelf, as a kind of VaaS (Verification-as-a-Service) platform. It is specially designed to deal with *certifier components*, a kind of components that can be connected to components of parallel computing systems for certifying them. The smooth orchestration of certifier components within parallel computing systems makes the proposed approach both *integrated*, in the sense that no disruptive elements must be considered in their architecture, and *modular*, as new certifier components may be included or released according to the sort of verification tasks required.

The certification framework also introduces the notion of *tactical components*, which help certifier components to access existing verification infrastructures, including theorem provers and model checkers. Tactical components, such as any components in HPC Shelf, are inherently parallel. So, they can be implemented using parallel programming techniques to exploit the maximum performance of the underlying cloud infrastructure during the certification process.

Another concept introduced in this paper is the *parallel certification system*. Analogous to parallel computing systems, in which a workflow component orchestrates a set of solution components over a set of virtual platforms in order to solve a problem, a parallel certification system contains a certifier component that orchestrates a set of tactical components according to the needs of the certification process required by a set of *certifiable components*.

As a case study, this paper introduces a particular family of certifier components, so-called C4 (Computation Component Certifier Components), for verifying functional and safety properties of computation components. In order to compose tactical components for C4 certifiers, we have studied the main verification techniques and associated tools that allow the verification of formal properties on the different classes of parallel programs. Finally, a specific C4 component and its tactical components have been applied to certify components of Montage (Berriman et al., 2004), an existing astrophotography software kit that we have been modeled as an HPC Shelf application.

This paper has six more sections. Section 2 describes HPC Shelf. Section 3 presents the current design of the certification framework. Section 4 presents the architecture of the computation certifiers (C4) and their tactical components. Section 5 contains the case study. Related works are presented in Section 6. Finally, Section 7 concludes this paper.

2 HPC Shelf

HPC Shelf is a cloud computing platform for developing and deploying cloud-based HPC applications that serve specialists of a given domain. For that, it provides, to application providers, tools for specifying problems and building the corresponding computational solutions by orchestrating components of different *component kinds*. Such computational solutions are called *parallel computing systems*, built out of components that address functional and non-functional concerns related to both hardware and software elements. For that, such components comply to Hash (de Carvalho Junior et al., 2007), a model of components intrinsically enabled for parallelism and supporting the notion of component kinds.

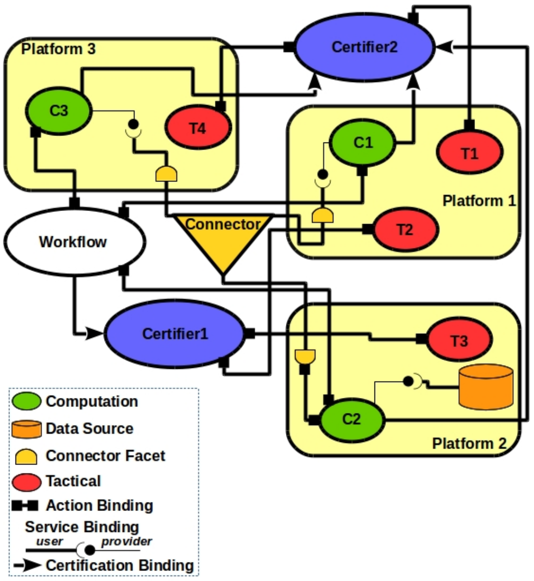

Figure 1: Components of a hypothetical parallel computing system with tactical and certifier components

2.1 Component Kinds of HPC Shelf

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

A *service binding* connects a *user* to a *provider* port, allowing a component to consume a service offered by another component. Such ports are compatible if a service binding exists for connecting them. In turn, *action bindings* connect a set of action ports that export the same set of action names. Two actions of the same name whose action ports are connected in two components execute when both components make them active at the same time (*rendezvous*). Figure 1 depicts a scenario illustrating components and their bindings, also including certifier and tactical components, proposed in this paper (Section 3).

Components have a predefined action port, called *LifeCycle*, with the following actions for controlling their life-cycle. Action **resolve** selects a component implementation and a virtual platform for it, according to a system of *contextual contracts* (Section 2.4). Action **deploy** deploys a selected component in a parallel computing platform. Action **instantiate** instantiates a deployed component, which becomes ready for communication with other components through service and action ports. Finally, action **release** releases a component from the platform on which it is instantiated, when it is no longer useful.

2.2 Stakeholders

The following stakeholders work around HPC Shelf. A ***specialist*** (final user) uses an application for specifying problems using a domain-specific interface and executing computational solutions built by the application. A ***provider*** creates and deploys applications, by designing computational solutions built out of components, in the form of parallel computing systems. A ***developer*** develops components of parallel computing systems, tuned for exploiting the architecture of a class of virtual platforms. For that, they are experts in parallel computer architectures and programming for them. A ***maintainer*** offers a parallel computing infrastructure, from which virtual platforms are instantiated.

2.3 Architecture

The architecture of HPC Shelf is based on the following three elements: Front-End, Core and Back-End.

The Front-End is SAFe (*Shelf Application Framework*) (de Carvalho Silva and de Carvalho Junior, 2016), a collection of Java classes and design patterns that *providers* use for deriving applications. It supports a language for specifying parallel computing systems, so-called SAFeSWL (SAFe Scientific Workflow Language). SAFeSWL is a language divided into an *architectural* and an *orchestration* subsets. The architectural subset is used to specify the components and bindings of a parallel computing system. In turn, the orchestration subset allows the definition of orchestration workflows of the parallel computing systems. Workflow components are in fact special connectors which run directly on SAFe.

The Core manages the life-cycle of components, from cataloging to deployment. Developers, certifiers and maintainers register components and their contracts through the Core. For that, the Core implements an underlying system of contextual contracts. Applications access the services of the Core for resolving component contracts and deploying the components of parallel computing systems.

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

2.4 Contextual Contracts

HPC Shelf employs a system of *contextual contracts* (de Carvalho Junior et al., 2016) that separates, for a component, its interface, so-called *abstract component*, from its implementation, so-called *concrete component*. So, one or more concrete components may exist in the Core's catalog for a given abstract component, to meet different assumptions about the requirements of the host application and the features of the parallel computing platforms where they can be instantiated (*context*). For that, an abstract component has a *contextual signature*, composed of a set of *context parameters*. In turn, a concrete component must be associated with a *contextual contract* that types it, defined by an abstract component and a set of *context arguments* that valuate its context parameters. During orchestration, when the action **resolve** of a component, specified by a contextual contract through SAFeSWL, is activated, a resolution procedure for its contextual contract is triggered for choosing of a specific concrete component that satisfies its contextual contract. Examples of contextual contracts will be presented in Section 4.2.

3 The Certification Framework

The certification framework herein proposed introduces a new kind of components, called *certifier components*, which makes it possible the certification of components of other kinds in a parallel computing system. The connection between a certifier component and a component to be certified is called *certification binding*, a new kind of binding, which joins service and action bindings. A component is considered *certifiable* if it is connected to one or more certifier components through certification bindings. A certifiable component has an additional action in its LifeCycle port, named **certify**. Another kind of components, called *tactical components*, is also proposed to meet the needs of certifier components to access the functionality of different verification infrastructures.

A new stakeholder, the ***certifier*** of components, which deals with certifier and tactical components, is proposed. They are specialists in formal methods and may be trained in parallel programming.

3.1 Parallel Certification System

A parallel certification system is composed of:

- a set of tactical components, executing in a predefined virtual platform where all artifacts needed by a given a proof infrastructure are deployed;
- a certifier component, responsible for orchestrating the tactical components in a certification task through a workflow written in TCOL (Tactical Component Orchestration Language);
- a set of certifiable components, linked to the certifier component through certification bindings.

Parallel certification systems are extracted from the SAFeSWL code of parallel computing systems. They are responsible for performing certification procedures on a set of components connected to the same certifier component. Note their analogy with parallel computing systems, where the certifier component is analogous to the workflow component and tactical components are analogous to solution components. Therefore, the certifier component is not associated with a virtual platform, running directly on SAFe.

3.2 Tactical Components

A tactical component represents a proof infrastructure, composed of a suitable combination of verification tools. So, it can perform a flow of execution that goes from receiving a code written in the language it understands, execute validations, conversions and, finally, verify properties on such a code. Tactical components may exploit the parallelism features of the virtual platforms where they are placed for accelerating verification.

A tactical component has the following ports:

- a user service port, with the following operations: receiving the code of the certifiable component from the certifier component, possibly previously translated by the certifier component into the language that the tactical component understands; receiving from the certifier component formal properties to be verified on that code; allowing the certifier component to monitor the progress of the verification of properties; and returning to the certifier the result of the verification process;
- an action port called *Verify*, containing the actions **verify_perform**, **verify_conclusive** and **verify_inconclusive**;
- the default port *LifeCycle*, which allows the certifier component to control its life-cycle.

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

3.3 Ports and Bindings of Certifiers

For a component to be certified, one or more certifier components must be chosen from the catalog and connected to it through certification bindings within the SAFeSWL architectural definition of the component. When the application's workflow activates the action **certify** of a certifiable component, a parallel certification system is instantiated and executed for each associated certifier component.

The certification of a component with respect to a certifier component is idempotent, that is, it is only executed once, even though the action **certify** is activated several times in one or more applications. Each certifier component differentiates which properties it verifies are mandatory and which are optional. At the end of the certification process, the component is considered to be certified with respect to the certifier component if all mandatory properties have been proven. If so, the component receives a *certificate* with respect to the certifier component, which is registered in the Core in an unforgeable format.

To communicate with the component to be certified, a certifier component has a unique provider service port with the following characteristics:

- through this port, SAFe passes, to the certifier, the component code, *ad hoc* formal properties (not generated by the certifier component itself nor annotated directly in the programs), if any, as well as other information relevant to the certification;
- through methods available on this port, the certification process of the component performed by the certifier component can be started by SAFe.

A certifier component has also ports that are counterparts of the verification ports of the associated tactical components. When SAFe requests the certifier component to start the certification process, the certifier component performs the orchestration defined in the associated TCOL fragment, which at some point will activate the action **verify_perform** related to each associated tactical component. If the action **verify_perform** is activated in a tactical component, its process of verification of formal properties is started. When a tactical component activates **verify_conclusive** and the action of the same name is activated in the certifier, the certifier accesses the service port of the tactical component in order to obtain the result of the verification of properties and make an accounting of which have been proven or not.

Finally, when **verify_inconclusive** is activated in the tactical component and the same occurs in the certifier, the certifier accesses the service port of the tactical component to obtain a report. If the certifier component has been implemented with this capability, it can request, either automatically or with the application permission, to the Core that, if possible, instantiate the same tactical component on another virtual platform. If this is possible, the certifier then restarts the verification process for the failed tactical component. If not, the certifier emits a message to the application and continues its normal execution flow, although considering the properties of responsibility of the failed tactical component as *inconclusive*.

3.4 Tactical Orchestration with TCOL

The automatic application of different tactical components to verify formal properties on a set of programs is designed to enhance the certification process. For example, a specific property that was not proved by a tactical component can be proved by others. Moreover, the parallel activation of tactical components may lead to a decrease in the final verification time.

Based on this scenario, we propose a language that allows the orchestration of a set of tactical components by a certifier component, called TCOL (Tactical Component Orchestration Language). In the current prototype, TCOL is an extension of the orchestration subset of SAFeSWL, which is efficient for orchestrating coarse-grained parallel components, such as tactical components. The extension was made by the addition of a new selection operator, **switch**. Here is the abstract syntax of TCOL:

TASK ::= **skip** | **perform** *action* | **start** *handle action* |
wait *handle* | **cancel** *handle* | **sequence** TASK+ |
parallel TASK+ | **repeat** TASK | **break** |
continue | **select** {*action*: TASK}+ |
switch {*condition*: TASK}+

An *action* can be an action of one of the predefined ports: *LifeCycle* and *Verify*. A task (TASK), consists of an orchestration of a set of actions. The orchestration performed by the certifier is defined by a higher-level task that orchestrates internal tasks. There are seven primitive tasks and five task combinators.

- **skip** denotes an empty action;
- **break** ends the iteration in which it is nested;
- **continue** forces the start of the next iteration;
- **perform** synchronously activates an action;
- **start** activates an action asynchronously, optionally handled by an identifier (*handle_id*);
- **wait** waits for the completion of an asynchronous action, possibly blocking the orchestration thread in which it was executed;
- **cancel** cancels the activation of an action previously activated asynchronously.

The task combinators are:

- **sequence** denotes the sequential execution of a list of tasks, in the order they are declared;
- **parallel** expresses the concurrent execution of a set of tasks, in which the combinator task ends after completion of these tasks (*fork-join* model);
- **repeat** represents the iterative execution of a task, in which this iteration is terminated when a **break** is executed within its scope;
- **select** selects one from a set of tasks, where chosen task is the first one in the sequence whose guard action is activated in the component;
- **switch** selects one from a set of tasks, where the selected task is the first one in the sequence whose condition is evaluated to true.

The condition grammar of TCOL is similar to the ones of C-like object-oriented languages. The current set of variables that may appear in conditions is managed by the certifier and they are updated by it during the certification process. They are:

- **formal_properties**: an associative array which associates each pair <*formal property, program*> to null, if no tactical component has tried or succeeded to verify (prove or disprove) the property; false, if the property was verified by some tactical component but was refuted; and true, if some tactical component succeeded to proved it;
- **tc_applied**: an associative array that indicates, for each tactical component orchestrated, whether it has already been activated to verify the formal properties assigned to it (false or true);
- **Context arguments**: it is possible to use in conditions the arguments assigned to the context parameters of the abstract certifier by the contextual contract of the concrete certifier.

4 The C4 Certifier

C4 (Computation Component Certifier Component) is a kind of *certifiers* aimed at verifying formal properties of computation components.

In HPC Shelf, computation components have a set of **units**, which represent processes that run on different processing nodes of the target virtual platform. Commonly, a component has a single *parallel unit*, representing a team of units programmed in SPMD (Single Program Multiple Data) style, where the same code is executed in each unit. The units work on different data partitions and synchronize by exchanging messages. This is the programming model of MPI (Dongarra et al., 1995). Parallelism in a unit may also be exploited by launching threads on the virtual platform node that places it by using OpenMP (OpenMP, 1997).

A computation component may also be constituted by multiple parallel units. Also, it may have singleton units, each one running a distinct program.This is the MPMD (Multiple Program Multiple Data) style.

Proving functional and safety properties of message-passing based parallel programs is considered a challenge task, even for SPMD.

4.1 Tactical Components for C4

We had performed a systematic survey for deciding which existing formal verification techniques and tools tactical components could support for verifying parallel programs in computation components. Our findings are summarized in the following sections, which discuss the two major classes of tools: *deductive program verification* and *model checking*.

4.1.1 Deductive Tactical Components

Tactical components for deductive verification are *assertional*, since programs they verify must be decorated with assertions of the Floyd-Hoare logic or its extensions, such as: *separation logic* (Reynolds, 2002), for mutable data structures; *Owicki-Gries reasoning* (Owicki and Gries, 1976), for shared-memory parallel programs; and *Apt's reasoning* (Apt, 1986), for distributed-memory ones.

A tactical component is *purely assertional* if the properties it verifies are only in the form of specification assertions, i.e., pre- and post-conditions of methods. In turn, a tactical component is *non-purely assertional* when the properties it verifies are *ad hoc*, i.e., properties created by the component developer and stored in the component to be verified on it.

For deductive verification tools, the verification time is not proportional to the number of units of the component. However, the application of these tools to distributed-memory parallel programs, especially MPI ones, is still incipient. In fact, we have found that only ParTypes (López et al., 2015) can verify C/MPI programs, annotated in the syntax of VCC (Cohen et al., 2009), against a high-level communication protocol, received by the certifier as an *ad hoc* property. With respect to thread-based programming in C and Java, explicit safety properties annotated in programs, as well as implicit ones (e.g. ensure that the program does not make access to unallocated memory locations), can be verified with VeriFast (Jacobs et al., 2011), provided that they are annotated with separation logic assertions. In turn, Frama-C (Cuoq et al., 2012) is a very expressive tool for verifying sequential C programs annotated as methods pre- and post-conditions.

VCC, VeriFast and Frama-C are *verification frontends*. They verify annotated programs written in a high-level language, generally translating them to an *intermediate verification language*, which generates verification conditions (VCs) and applies them to an automatic or interactive prover. Intermediate verification languages act as layers upon which verifiers are built for other languages. Boogie (Barnett et al., 2006) and Why3 (Bobot et al., 2011) are examples of them.

There is a wide variety of automatic provers, from two main classes: SMT (Satisfiability Modulo Theories) solvers and ATP (Automated Theorem Provers) provers. SMT solvers determine when formulas in first order logic, in which some symbols of functions and predicates have additional interpretations, are satisfiable. Among them, we can mention Alt-Ergo, CVC3, CVC4 and Z3. ATP provers, in turn, deal with the task of proving that a conjecture is a logical consequence of a set of axioms and hypotheses. They include E, SPASS and Vampire. There is also a prover for verifying floating-point arithmetic, called Gappa.

In general, a deductive verification tactical component 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, may add new features.

When there is a single possible composition, the tactical component is named with the most specialized tool. For example, ParTypes is composed of ParTypes, VCC, Boogie, and Z3. In turn, when there are multiple ones, the name is composed of excerpts from the names of the tools. For example, JFWCVC4 is composed of Jessie, Frama-C, Why3, and CVC4.

4.1.2 Model Checking Tactical Components

The scarcity of deductive verification tools for MPI programs has motivated us to investigate model checking alternatives. The most relevant we have found are ISP (In-situ Partial Order) (Vo et al., 2009) and CIVL (Concurrency Intermediate Verification Language) (Siegel et al., 2015). They verify a fixed, although expressive, set of safety properties. ISP verifies properties like deadlock absence, assertion violations, MPI object leaks, and communication races (unexpected communication matches) on C, C++ and C# programs carrying MPI/OpenMP directives. CIVL, in turn, includes the verification of functional equivalence between programs.

All standard properties of a tactical component are mandatory. For a specification assertion to be considered mandatory, it must be preceded by the annotation @*Mandatory*. Otherwise, it is optional. Finally, with respect to *ad hoc* properties, the component developer must make the distinction. Examples of standard and *ad hoc* properties are present in the case study.

4.2 Contextual Contracts

In general, computation certifiers differ from one another by the set of tactical components they are able to orchestrate. Therefore, contextual contracts of these components must express characteristics of their associated tactical components.

Each certifier keeps track of the average verification time of each property it verifies, keeping this information in the component description in the catalog of components. Thus, the component developer can evaluate the cost/benefit of verifying a particular property. Moreover, it may be interesting to her to establish a maximum time that she is willing to dispense with the verification of all formal properties on her component. Thus, there is a *quality* context parameter in the certifier that must be valued with this time and the Core will only choose certifiers that are able to verify the properties within that time. The initial maximum verification time can be obtained by analytical modeling or experimentation by the certifier developer. The Core keeps track of the occurrences of violations in quality parameters, including verification time of certifiers. Such a kind of *reputation* parameter influences the Core in determining the priority of components (certifiers) among the possible choices calculated by the resolution procedure. However, a description of such a mechanism is out of the scope of this paper.

The abstract C4 certifier has currently the following contextual signature:

$$\mathrm{C4}\ \left[\begin{array}{l} \textbf{\textit{programming\_language}} = P : \textsc{PLType}, \\ \textbf{\textit{separation\_logic}} = S : \textsc{SLType}, \\ \textbf{\textit{floating\_point\_operations}} = F : \textsc{FPOType}, \\ \textbf{\textit{existential\_quantifier}} = E : \textsc{EQType}, \\ \textbf{\textit{message\_passing\_interface\_verif}} = M : \textsc{MPIVerifType}, \\ \textbf{\textit{ad\_hoc\_properties}} = A : \textsc{AHType}, \\ \textbf{\textit{max\_verification\_time}} = MV : \textsc{Integer} \end{array}\right]$$

Here is the description of the context parameters:

- ***programming_language***: denotes the programming languages in which programs that the certifier verify must be written;
- ***separation_logic***: states whether the certifier orchestrates tactical components able to verify programs annotated with separation logic assertions;
- ***floating_point_operations***: indicates whether the certifier is able to verify floating point operations, by using the tactical component Gappa;
- ***existential_quantifier***: indicates whether the assertions of programs to be verified by the certifier contain existential quantifiers. SMT solvers are not good at guessing witnesses for existentially quantified variables (François Bobot et al., 2014). In this case, if the certifier orchestrates a set of SMT solvers and ATP provers, it is advisable that it first invoke the ATP provers;
- ***message_passing_interface_verif***: states whether that the certifier is able to verify MPI programs;
- ***ad_hoc_properties***: informs whether the certifier accepts *ad hoc* properties;
- ***max_verification_time***: denotes the certifier maximum verification time (seconds).

PLTYPE, SLTYPE, FPOTYPE, EQTYPE, MPIVERIFTYPE, AHTYPE and INTEGER are abstract qualifier components that determine the contractual bounds of the context parameters.

An example of contextual contract derived from the contextual signature of C4 is the contextual contract of the concrete certifier C4ImplMo, which will be better contextualized in the case study:

$$\mathsf{C4ImplMo} : \mathrm{C4}\ \left[\begin{array}{l} \textbf{\textit{programming\_language}} = \mathrm{C}, \\ \textbf{\textit{separation\_logic}} = \textsc{NoSeparationLogic}, \\ \textbf{\textit{floating\_point\_operations}} = \textsc{NoFloatingPointOperations}, \\ \textbf{\textit{existential\_quantifier}} = \textsc{NoExistentialQuantifier}, \\ \textbf{\textit{message\_passing\_interface\_verif}} = \textsc{MPIVerif}, \\ \textbf{\textit{ad\_hoc\_properties}} = \textsc{AdHocProperties}, \\ \textbf{\textit{max\_verification\_time}} = 30 \end{array}\right]$$

Based on this contract, we can state that C4ImplMo verifies C/MPI programs, accepts *ad hoc* properties and its initial (experimental) maximum verification time is 30 seconds.

C4ImplMo orchestrates two tactical components, a model checking one (ISP) and a deductive verification one (PARTYPES), which extend the following contextual signature with no addition of parameters:

$$\textsc{Tactical}\ \left[\begin{array}{l} \textbf{\textit{message\_passing\_interface}} = M : \textsc{MPIType}, \\ \textbf{\textit{number\_of\_nodes}} = N : \textsc{Integer}, \\ \textbf{\textit{number\_of\_cores}} = C : \textsc{Integer} \end{array}\right]$$

```
<sequence>
    <parallel>
        <sequence>
            <perform action="resolve" comp="ISP"/>
            <perform action="deploy" comp="ISP"/>
            <perform action="instantiate" comp="ISP"/>
        </sequence>
        <sequence>
            <perform action="resolve" comp="ParTypes"/>
            <perform action="deploy" comp="ParTypes"/>
            <perform action="instantiate" comp="ParTypes"/>
        </sequence>
    </parallel>
    <parallel>
        <perform action="verify_perform" comp="ISP"/>
        <perform action="verify_perform" comp="ParTypes"/>
    </parallel>
    <perform action="release" comp="ISP"/>
    <perform action="release" comp="ParTypes"/>
</sequence>
```

Figure 2: The orchestration of C4ImplMo in TCOL

The parameter ***message_passing_interface*** states whether the tactical component is implemented through the MPI library (note the difference between this parameter and ***message_passing_interface_verif*** of C4). The parameters ***number_of_nodes*** and ***number_of_cores*** mean, respectively, the minimum number of processing nodes and processing cores per processing node that the target platform of the tactical component must contain. The orchestration performed by C4ImplMo is governed by the TCOL fragment in Figure 2.

As concrete tactical components for the ones described above, we cite, respectively, ISPImpl and ParTypesImpl, which were implemented through the MPICH2 library [1] and require at least 4 processing cores in the target virtual platform:

$$\text{ISPImpl} : \text{ISP}\ [\textbf{\textit{message\_passing\_interface}} = \text{MPICH2},\ \textbf{\textit{number\_cores}} = 4]$$

$$\text{ParTypesImpl} : \text{P}\textsc{ar}\text{T}\textsc{ypes}\ [\ \textbf{\textit{message\_passing\_interface}} = \text{MPICH2},\ \textbf{\textit{number\_cores}} = 4]$$

In C4ImplMo, the valuations to the contextual signatures of the associated abstract tactical components were the following, thus making ISPImpl and ParTypesImpl possible candidates returned by the Core, by considering that MPI is a supertype for MPICH2:

$$\text{ISP}\ [\textbf{\textit{message\_passing\_interface}} = \text{MPI}]$$

$$\text{P}\textsc{ar}\text{T}\textsc{ypes}\ [\textbf{\textit{message\_passing\_interface}} = \text{MPI}]$$

Suppose that in order to make a specific computation component certifiable, its developer has created a certification binding between it and C4, for which the following contextual contract has been provided:

$$\text{C4}\ \begin{bmatrix} \textbf{\textit{programming\_language}} = \text{C}, \\ \textbf{\textit{ad\_hoc\_properties}} = \text{A}\textsc{d}\text{H}\textsc{oc}\text{P}\textsc{roperties}, \\ \textbf{\textit{message\_passing\_interface\_verif}} = \text{MPIV}\textsc{erif} \end{bmatrix}$$

This valuation states that she wants the component to be certified by a certifier that verifies C/MPI programs and accepts external properties (*ad hoc*). Clearly, C4ImplMo is a candidate for this valuation.

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

5 Case Study: Montage

Montage (Mosaic Astronomic Engine) (Berriman et al., 2004) is an astrophotography software toolkit for composing astronomical mosaics (sets of images of a specific area) that preserve the calibration and the position of the original input images and may represent areas of the sky that are too big to be played by astronomical cameras. Montage is used as a case study in several scientific workflow research projects.

The toolset has a number of independent and self-executable components that can be orchestrated for obtaining the mosaic. In general, each component takes as input a set of files and input parameters, and outputs files (possibly images) for other components. We have encapsulated them as computation components of HPC Shelf. For that, they provide a single computational action, written in C. Those parallel ones use MPI for simulating the parallel execution of instances of another associated sequential component, each one working on distinct input files.

We illustrate the use of Montage components with an existing workflow that generates a mosaic of the Pleiades star cluster[2], with the following components:

- MARCHIVELIST: given a sky coordinate and an archive name, it retrieves, from IRSA (Infrared Science Archive)[3], a list of images that overlap that position and stores them into the archive;
- MARCHIVEEXEC: retrieves from the server the FITS (Flexible Image Transport System)[4] images of each image listed in an archive and stores them into the current directory;
- MIMGTBL: generates metadata in a table from images contained in a directory;
- MPROJEXEC: reprojects images contained in a directory, by using information contained in the metadata table. In fact, it executes a set of instances of the component MPROJECT in parallel. Each one reprojects one or more images;
- MADD: coadds all images previously corrected into the final image;
- MJPEG: generates a JPEG file for the final image.

For each image that overlaps the center of the Pleiades cluster, the workflow keeps three versions of it, each one in a different color band (red, infrared and blue). The images of the same band are stored, respectively, in the data source components DSS2RDIR, DSS2IRDIR and DSS2BDIR. The color bands are processed by separate parallel sub-workflows. At the

[2] http://montage.ipac.caltech.edu/docs/pleiades_tutorial.html

[3] http://irsa.ipac.caltech.edu/ibe

[4] http://fits.gsfc.nasa.gov

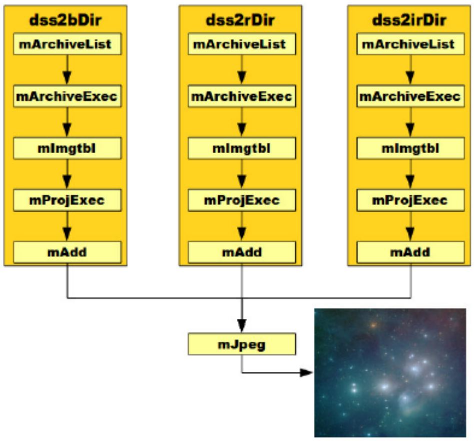

Figure 3: The three sub-workflows of the Pleiades workflow. At the end, the images of each color band are combined by the component MJPEG.

end, the images of each color band are overlapped, thus generating the final image. Figure 3 shows the orchestration logic of the computation components of the Pleiades workflow. Due to the space restrictions, we have omitted, the contextual signatures of the abstract components, the contextual contracts of the concrete components, and the workflow code.

5.1 Certifying Montage Components

The highest processing load in Montage falls on the parallel components, since their associated sequential components perform the most critical computational actions. Hence, our efforts focused on certifying these components. They are: MADDEXEC, MBGEXEC, MDIFFEXEC, MFITEXEC and MPROJEXEC. They have a single parallel unit, running a SPMD program.

In order to certify the Montage parallel components in batch mode, the developer may create through SAFe a *certification application*, i.e. dedicated to certify components, by creating a parallel computing system containing the components to be certified and the associated workflow, described in the SAFeSWL fragment in Figure 4.

```
<parallel
    <sequence>
        <perform action="resolve" comp="mAddExec"/>
        <perform action="certify" comp="mAddExec"/>
    </sequence>
    <sequence>
        <perform action="resolve" comp="mBgExec"/>
        <perform action="certify" comp="mBgExec"/>
    </sequence>
    <sequence>
        <perform action="resolve" comp="mDiffExec"/>
        <perform action="certify" comp="mDiffExec"/>
    </sequence>
    <sequence>
        <perform action="resolve" comp="mFitExec"/>
        <perform action="certify" comp="mFitExec"/>
    </sequence>
    <sequence>
        <perform action="resolve" comp="mProjExec"/>
        <perform action="certify" comp="mProjExec"/>
    </sequence>
</parallel>
```

Figure 4: SAFeSWL code for orchestrating the Montage parallel components in the certification application

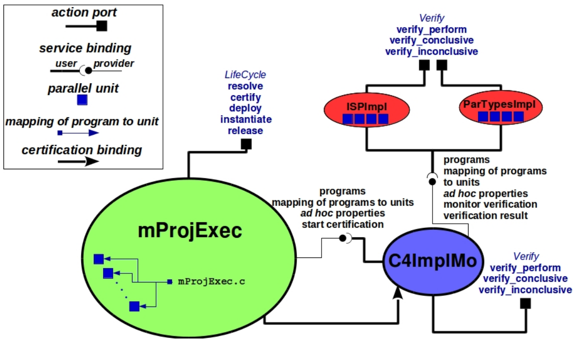

Figure 5: Certification architecture of MPROJEXEC

By assuming that certification bindings between these components and C4 have been created, as described in Section 4.2, the developer creates an appropriate service binding for information exchange between each one and C4, fixes the choice of C4ImplMo for each one, sets up the single program of each Montage parallel component to be verified by both tactical components of C4ImplMo, ISP and PARTYPES, and records on each component, as an *ad hoc* property, a ParTypes protocol to be verified against the implementation of its program. Finally, the developer requests the execution of the certification application. SAFe then creates automatically the certification system. Upon the activation of the action **certify** of a component in the certification workflow, the component joins the parallel certification system to be certified.

The architectural certification scenario concerning to each Montage parallel component is similar and we particularize, for didactic purposes, for the component MPROJEXEC, in Figure 5. For all components, virtual platforms containing 20 processing nodes were chosen. For the tactical components of C4ImplMo, respectively, ISPImpl and ParTypesImpl were chosen, for which virtual platforms with 4 processing node were chosen. As a ParTypes protocol to be verified against the program of MPROJEXEC, `mProjExec.c`, the protocol in Figure 6 was provided in this component. This protocol states that `mProjExec.c` executes a sequence of eight **MPI_Allreduce** operations, each of which applies a reduction (in this case, sum) of integers from all processes and distributes the result back to all processes.

Operationally, when the orchestration concerning to mProjExec in the parallel certification system terminates, the vector **formal_properties**, maintained by the certifier component, has the result of the verification of all properties, therefore considering mProjExec certified with respect to C4ImplMo:

- **formal_properties**[“mProjExec.c” , “no deadlock”] = true
- **formal_properties**[“mProjExec.c” , “no MPI object leaks”] = true
- **formal_properties**[“mProjExec.c” , “no communication races”] = true
- **formal_properties**[“mProjExec.c” , “no irrelevant barriers”] = true
- **formal_properties**[“mProjExec.c” , “protocol mProjExec”] = true

All parallel components except mBgExec were certified. Indeed, ISP has detected a possible interleaving in mBgExec that can lead to receiving an empty buffer in a **MPI_Allreduce** operation.

```
protocol mProjExec { allreduce sum integer[1]
                     allreduce sum integer[1]
                     allreduce sum integer[1]
                     allreduce sum integer[1]
                     allreduce sum integer[1]
                     allreduce sum integer[1]
                     allreduce sum integer[1]
                     allreduce sum integer[1] }
```

Figure 6: ParTypes protocol to be verified against the program `mProjExec.c`, annotated with VCC assertions

6 RELATED WORK

Verification as a Service (VaaS) has been introduced by (Schaefer and Sauer, 2011) to deal with scale problems of software formal verification in large-scale software systems. It is based on verification workflows, which can run on service-oriented computing environments, reducing the complexity of verification services. Starting from a system and properties to be verified, a verification workflow can be performed by invoking verification tasks in an appropriate order. The authors argue that cloud computing, by providing a service-oriented environment and an extremely powerful shared processing infrastructure through an abstract interface, becomes the best alternative to implement VaaS architectures.

Only a few research initiatives on VaaS exist. In (Mancini et al., 2015), a verification service is proposed to show system correctness with respect to uncontrollable events through an exhaustive hardware simulation by considering all relevant scenarios. In (Bellettini et al., 2015), a Map-Reduce algorithm for checking CTL formulas on a cloud is proposed.

The closest initiative to our work is the framework proposed by Hu *et al* (Kai Hu et al., 2016). They propose a robust VaaS framework, focusing mainly on the dualism with the main concerns of SaaS (Software-as-a-Service), such as the storage of verification tools and results, scalability issues and fault tolerance. Their approach, however, allows the availability of verification tools in the cloud in a raw way, thus requiring an application developer with skills in the kind of verification supported by the tool, which it is not often true. Also, their approach does not make it possible to make assumptions about the computing platforms on which the verifications will be performed. Indeed, the use of parallel computing platforms for accelerating verifications is not a concern.

This paper introduces the first VaaS framework aimed at HPC requirements. Also, it introduces innovative features compared to the framework of Hu *et al.* For instance, application developers only select certifier components for certifying components they desire, and the entire certification process runs automatically, by orchestrating tactical components in a prescribed workflow. Moreover, certifier and tactical components make use of HPC techniques, being able to exploit different levels of parallelism supported by HPC platforms (distributed-memory, shared-memory, multi/many-core, accelerators, etc). Indeed, accelerating component certification during orchestration is a relevant concern since it is a prerequisite for running certifiable components in an application.

7 Conclusions and Future Work

This paper introduced a VaaS (Verification-as-a-Service) framework on top of the HPC Shelf platform, thus attending requirements of component-based high performance computing (CBHPC) platforms.

Such a framework aims at attesting the reliability of components, w.r.t. both malicious behavior of third parties codes and anomalous or erroneous behavior caused by programming errors. Such errors are potentially more common in parallel components of HPC Shelf, possibly invalidating scientific studies, due to the production of results that are erroneous or whose reliability cannot be proven, as well as waste precious time in long-running computations.

In order to make the process of certifying components less disruptive as possible, the concept of *parallel certification system*, built of components, was introduced for reusing the workflow already used by the application provider to build parallel computing systems. In fact, component orchestrations in both kinds of systems are operationally similar and can be overlapped without interference during execution.

The certification framework is an ongoing project whose initial prototype has been developed in C#/MPI and validated through the Montage application. These implementations can be found at `https://github.com/UFC-MDCC-HPC/HPC-Shelf-Certification`.

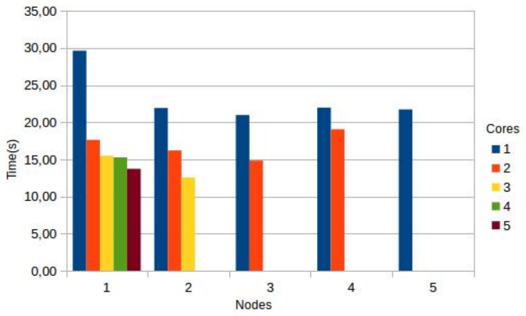

Figure 7: Execution times of the certification application

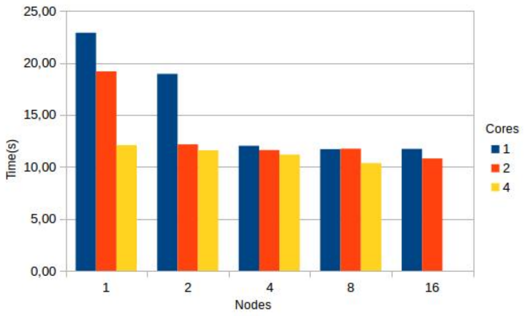

Figure 8: Map-Reduce workflow certification times

Figure 7 shows the experimental time for executing the certification application of the case study, by varying the number of processing nodes and processing cores per node engaged in the execution of the tactical components. The sequential time was calculated outside HPC Shelf, simply by sequential calls to the verification tools through a Shell Script program. The parallelism was justified in all cases, leading in some cases to a certification time of less than half the sequential time. Such a gain can be useful to the application provider, since the certification of certifiable components is a prerequisite for running them. We think that this gain could become even more evident if components have more programs to be certified or the certifier can orchestrate more tactical components.

An outstanding feature of the proposed certification framework is *extensibility*, since new certifier and tactical components may be smoothly added to deal with the verification of different component kinds and different classes of properties as verification tools evolve. In particular, we intend to continue the investigation about verification tools for MPI programs, since MPI is a *de facto* standard in HPC.

We are also working on SWC2 (Scientific Workflow Certifier Component), a certifier for certifying workflows of parallel computing systems of HPC Shelf w.r.t. behavioral properties (safety and liveness). The architecture of such a component and the formal modeling of the workflows it performs to the mCRL2 (Groote et al., 2007) verification toolset are aimed at enriching the formal specifications of these components in order to increase their confidence levels (not only for the detection of design errors on them) and will be presented in a separate paper. An implementation of such a component on top of the certification framework was evaluated through the certification of the Map-Reduce (Dean and Ghemawat, 2008) processing workflow and reveals promising results, as can be seen in Figure 8.

We also argue that the reliability requirements of HPC Shelf components may be extrapolated to other component-based service-oriented HPC platforms. In particular, our immediate interest is to port the workflow certification service to other workflow-based platforms, since the verifiable workflows patterns in SWC2 are common to most of them.

From a broader perspective, we believe that the inclusion, at design time of an application, of components that in a reflexive way are able to inspect and certify components is worth exploring for two reasons. On the one hand, the inherent complexity of cloud-based applications, given the heterogeneity of resources and the open and decentralized control they have, implies the need to scale up modeling and 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, as suggested in this paper. While the first perspective has already been the focus of a number of research initiatives (see, for example, the ABS project (Albert et al., 2014)), the second one constitutes an open challenge to the software engineering community. The HPC Shelf platform with reflexive certification of components enabled, as introduced in this paper, is a step in this direction.

ACKNOWLEDGEMENTS

This paper is a result of the project SmartEGOV: Harnessing EGOV for Smart Governance (Foundations, methods, Tools) / NORTE-01-0145-FEDER-000037, supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (EFDR).

REFERENCES

Albert, E., de Boer, F. S., Hähnle, R., Johnsen, E. B., Schlatte, R., Tarifa, S., and Wong, P. (2014). Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications*, 8(4):323–339.

Apt, K. R. (1986). Correctness Proofs of Distributed Termination Algorithms. *ACM Trans. Program. Lang. Syst.*, 8(3):388–405.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). 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, pages 364–387. Springer-Verlag.

Bellettini, C., Camilli, M., Capra, L., and Monga, M. (2015). Distributed CTL model checking using MapReduce: theory and practice. *Concurrency and Computation: Practice and Experience*.

Berriman, G. B., Deelman, E., Good, J. C., Jacob, J. C., Katz, D. S., Kesselman, C., Laity, A. C., Prince, T. A., Singh, G., and Su, M.-H. (2004). Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *SPIE Astronomical Telescopes+ Instrumentation*, pages 221–232. International Society for Optics and Photonics.

Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. (2011). Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64.

Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. (2009). VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer.

de Carvalho Junior, F. H., Lins, R., Correa, R. C., and Araújo, G. A. (2007). Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience*, 19(5):697–719.

de Carvalho Junior, F. H., Rezende, C. A., Silva, J. C., and Al Alam, W. G. (2016). Contextual abstraction in a type system for component-based high performance computing platforms. *Science of Computer Programming*.

de Carvalho Silva, J. and de Carvalho Junior, F. H. C. (2016). A Platform of Scientific Workflows for Orchestration of Parallel Components in a Cloud of High Performance Computing Applications. In *Lecture Notes in Computer Science*. Springer.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dongarra, J., Otto, S. W., Snir, M., and Walker, D. (1995). An Introduction to the MPI Standard. Technical Report CS-95-274, University of Tennessee.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich (2014). Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–19.

Groote, J. F., Mathijssen, A., Reniers, M., Usenko, Y., and van Weerdenburg, M. (2007). The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems: Dagstuhl Seminar 06351*.

Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. (2011). VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55. Springer.

Kai Hu, Lei Lei, and Wei-Tek Tsai (2016). Multi-tenant Verification-as-a-Service (VaaS) in a cloud. *Simulation Modelling Practice and Theory*, 60:122 – 143.

López, H. A., Marques, E. R. B., Martins, F., Ng, N., Santos, C., Vasconcelos, V. T., and Yoshida, N. (2015). 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*, pages 280–298. ACM.

Mancini, T., Mari, F., Massini, A., Melatti, I., and Tronci, E. (2015). SyLVaaS: System level formal verification as a service. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 476–483. IEEE.

OpenMP, O. (1997). A Proposed Industry Standard API for Shared Memory Programming. *OpenMP Architecture Review Board*, 27.

Owicki, S. and Gries, D. (1976). An axiomatic proof technique for parallel programs I. *Acta informatica*, 6(4):319–340.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE.

Schaefer, I. and Sauer, T. (2011). Towards verification as a service. In *International Workshop on Eternal Systems*, pages 16–24. Springer.

Siegel, S. F., Zheng, M., Luo, Z., Zirkel, T. K., Marianiello, A. V., Edenhofner, J. G., Dwyer, M. B., and Rogers, M. S. (2015). CIVL: the concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 61. ACM.

Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. (2009). Formal Verification of Practical MPI Programs. *SIGPLAN Not.*, 44(4):261–270.

View publication stats