2011 IEEE Asia -Pacific Services Computing Conference

Elastic High Performance Applications – A Composition Framework

Tran Vu Pham
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology
t.v.pham@cse.hcmut.edu.vn

Hong-Linh Truong, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
{*truong,dustdar*}*@infosys.tuwien.ac.at*

Abstract—**With diverse and rich offerings from cloud computing providers in the open cloud market, scientists have great opportunities to design and conduct complex applications by utilizing and combining computational resources, software components and data sources in elastic manners. While existing techniques focus mainly on resource elasticity in single cloud infrastructure, scientists expect to design their applications being elastic in multiple dimensions to ensure that they applications can operate on multiple clouds with minimum software engineering effort. In this paper we will focus on providing techniques for scientists to compose elastic high performance applications by utilizing traditional software components, user-provided components and cloud services. We characterize elastic compositions via their resource, quality, cost, available time and usage right elasticity, thus enabling scientists to evaluate and decide how to develop, deploy and control the compositions to match their elastic needs. To illustrate our approach, we will present several real-world application compositions for multi-cloud environments.**

I. Introduction

Given complex computational models, scientists need to gather and assembly computational resources and software components to model and execute them. In conventional high performance computing (HPC) environments, typically scientists can (i) select, buy and deploy existing (licensed)software applications, and (ii) develop, select and buy software components and build customized applications by combining existing self-developed and free/bought software components.

With the emerging cloud computing paradigm, in addition to the above methods, scientists can also subscribe to cloud services (i.e. Software-as-a-Service (SaaS), Infrastructure-as-a-Service (IaaS)) or lease different components, such as applications, middleware, operating systems, and hardware infrastructures, and then compose them together with bought, free or owned components to meet their particular needs. Depending on different factors, with the same application, some scientists may want to be delivered at the very good quality in a very short time, while others may prefer to use it with a minimal cost, regardless of service quality and time. In short, requirements from scientists are *elastic* in different dimensions, including resource, quality, cost, available time and usage rights.

Currently, efforts for HPC application development have been devoted for moving existing applications onto clouds. However, for scientists, who do not have dedicated systems, support for application composition process to meet their requirements is crucial: applications should be utilized whatever exist in traditional software components, cloud services as well as user-provided components, and should be executed in suitable, possible cross-provider, private computing and public cloud systems. However, it is challenging to realize this for the HPC domain in which complex application compositions are typically done manual by scientists and various market factors, such as software cost and licensing, are not well examined.

In this paper, we introduce a framework for composing high performance applications from existing hardware and software components and available services in cloud environments to meet the various requirements from scientists. In addition to functional constraints, the composition framework will also take into account different elastic factors in order to flexibly meet scientists' demands. The composed applications can be deployed and executed in clouds. They can be used for modeling internal structure of applications so that we can make decision on which software components should be developed, bought or rented in order to extend/move existing applications into multi-cloud environments. We refer to these newly composed and deployed applications as elastic high performance applications (*eHPAs*), which are characterized by multi-elasticity dimensions, including resource, cost, quality, available time and usage rights. In order to achieve this objective, the framework addresses two important issues: (i) modeling various components that exist in HPC cloud computing environments and their elastic properties. (ii) composing the various available components into *eHPAs* that satisfy elastic user requirements.

The rest of this paper is organized as follows. Section II presents the background, motivation and related work. In Section III we conceptualize our elastic components and *eHPA* models. We describe our prototype in Section IV and experiment in Section V. Section VI concludes the paper and outlines our future work.

The work mentioned in this paper is partially funded by TRIG Program of Vietnam National University - Ho Chi Minh City.

II. Motivation and Related Work

A. Motivation

Cloud computing opens up many different ways for the users to access HPC capabilities. Various stakeholders are involved in a HPC cloud market environment. *Infrastructure Providers* (e.g. Amazon EC2) provide computational resources (RAM, CPU cycles, storage, networks, etc.) in form of IaaS and computational platforms as Platform-as-a-Service (PaaS). *Software Providers* can contribute to the environment software components in binary/source code formats or in form of SaaS. *Service Vendors* can utilize available resources (e.g. code, software, SaaS, PaaS and IaaS) on the environment, compose them and deliver the composed functionalities as SaaSs. *End Users*, in addition to consuming the products provided by other stakeholders, can also provide their own code/prototypes to the environment. Ideally, in HPC cloud environments, End Users should be able to select various components, e.g. HPC programs, libraries, operating systems, virtual machine images, Web Services, SaaS, PaaS and IaaS, and compose them together, deploy and run on cloud infrastructures to achieve the desired HPC capability.

With the offerings in the HPC cloud market, scientists – End Users in our work – have great opportunities to scale in/out their complex experiments using resources available in HPC cloud environments. As common in scientific research, the scientists may find and compose different components for their applications, and, if needed, to develop missing components for the applications. Then, they can deploy the application package on on-premise or public cloud computing infrastructures. Based on the deployment, different experiments could be defined and executed.

Composing such complex applications in clouds needs to take into account the scientists' different elastic requirements. They need some specific functions to be performed at certain quality. However, they are constrained elastically by time and money, and hence the HPC resources that they can access. Within these constraints, the composed applications should be able to scale in and out across different executing environments (e.g. on-premise and public clouds) as smooth as possible, so that scientists can take the most advantages from the resources they have access to, while minimizing efforts required for rebuilding or developing new components. To this end, we need:

- to solve complex system dependencies and conflicts for HPC software components in multiple execution environments as applications do not follow a single execution environment or programming model.
- to characterize elastic properties and to determine and associate these properties with applications explicitly in order to support the evaluation and control of elasticity of applications across multiple execution environments.

Resolving system dependencies and conflicts in HPC is challenging. It gets more complicated when there is a mixture of different software components and cloud services involved. Solutions for resolving software dependency and compatibility have been implemented in traditional software management systems (e.g. Linux yum and RPM), but they only deal with the software dependencies and compatibilities centered around a fixed operating system. They cannot be applied to the above scenario, where the focus is on applications. Dependency resolutions also exist in workflow composition systems, but they are often limited to the matching of service inputs and outputs. In HPC, an *eHPA* may span on multiple computer systems (i.e. MPI programs). Therefore, the dependencies and conflicts exist between not only components within a computing system but also components of different systems. Secondly, with respect to elasticity properties, current existing solutions on workflow or service compositions deal mainly with resources. Some solutions take into account of cost and quality of services. However, these properties are considered as fixed (non-elastic) properties. In our case, we will deal with elasticity in many different dimensions, including cost, quality, resource, available time and rights.

B. Related Work

Our work focuses on building *eHPAs* atop clouds, by composing available hardware, software and services available in cloud environments. Although similar, but it is not a workflow-based composition. It is also different from building and resolving software dependencies in traditional HPC applications, where everything happens in a specific environment. Therefore, existing scientific workflow composition techniques [1], [2], high performance application component models [3], or wrapping scientific applications for cloud environments [4] are not adequate.

To some extent, the software composition model for *eHPAs* introduced in this paper is similar to that of software product lines [5]. However, unlike software product lines, in which software components are owned by the organization/people whole run the product lines, in our models, software components come from many different sources including cloud services. This leads to several new constraints. Relating to software product line engineering (SPLE), non-functional aspects (performance, licensing etc.) are also considered. However, one of the major limited features is that they do not support elasticity of software components in the composition processes. Typically, SPLE supports only a set of fixed, known software in-house components. Thus, it also does not deal with issues related to licensing and multiple cost models. In Web services composition, several non-functional approaches exist but they focus only on the service model.

Scientific workflows and service composition support the composition of different software and services [1], [2], [6]. In existing workflow systems, they mostly support the composition of applications and services which have been deployed with clear functionality. Typically, workflows assume that applications and services to be composed are readily. In our work, we build different types of software components considering their system dependencies and conflicts. Furthermore, we consider software components at different levels, such as applications, libraries, middleware, and operating systems. Workflow composition techniques do not consider software at multiple levels of abstractions.

Techniques to model variability in existing tools [7] do not consider rich elastic properties in *eHPAs*, for example, cost, quality, available time and usage right. Several tools have been proposed for building and deploying multi-tenant applications [8] in business domain and they also consider different dynamic properties. However, *eHPAs* are much more complex than these multi-tenant applications with respect to software component types and execution environments. In our work, we focus on composing and modeling *eHPAs* which can be passed to other tools to build and move *eHPAs* to clouds. Therefore, existing package tools [9] and multi-cloud execution frameworks [10] are complementary to our work.

III. Elastic High Performance Application Models

A. eHPA Component Model

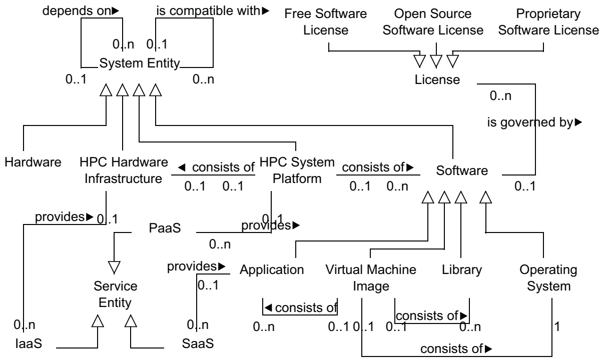


Figure 1. eHPA components and their inter-relationships

Figure 1 conceptualizes various HPC components and their inter-relationships, with respect to the HPC open market discussed in Section II. Two groups of entities exist in the model: system and service entities. System entities are those that reside in computer systems. In particular, System entities are grouped into the following categories:

- *Software:* refers to all software components. It is further classified into Application Software, Middleware and System Software. Application software delivers directly some capability to End Users. such as visualization. Middleware refers to third party libraries or shared APIs. System Software is used to describe operating systems and system software such as drivers.
- *Hardware:* is used to describe computing hardware in general, such as type of CPU and storage.
- *Software Platform:* refers to platforms on which software applications can run. A platform generally consists of an operating system and some common middleware in an integrated view. The entire software platform may be packed in a virtual machine image. It is typical in cloud computing environment that users may request for a virtual machine, then load a desired virtual machine image to set up the software platform.
- *Hardware Infrastructure:* refers to complete sets of computer hardware such as CPU, memory, storage and network. Hardware infrastructures may be virtually provided in form of IaaS.
- *HPC Compute Node:* for describing a compute node in an HPC systems, e.g. a node in a cluster.
- *HPC System Platform:* is used to describe a complete HPC systems, such as a cluster. An HPC system platform consists a software platform and hardware infrastructure. If the HPC system is a cluster, it may consists of many HPC compute nodes.

We use service entities to represent services available in clouds, such as IaaS, SaaS and PaaS. As in cloud computing service model, system entities can be provided via service entities.

Components in the above-mentioned categories can depend on or conflict with each others. Furthermore, components having similar attributes can be grouped into a *Type*, which is useful in case where a component depends on a type of component instead of any specific component of that type. In cloud-based HPC, an *eHPA* will consist of entities that will be executed in cloud-based HPC System Platforms.

B. Component Elasticity

In addition to the functional apsects, such as dependency and conflict, each component is also characterized by a set of non-functional properties including resoures, cost, quality, available time and usage rights. Similar to user requirements, these properties are also elastic. For example, in terms of resource, an parallel application may only function properly within a specific range of processes. Its cost may also be elastic according to the actual number of processes it uses. Table I shows a summary of these elastic properties and respective requirements from End User/Service Vendor and Software/Infrastructure Providers.

Due to elastic properties, discovering and matching user requests to components *eHPAs* during composition process are not only performed on the functional properties but also on elastic properties of the request and components.

Properties	**Description**	**End User/Service Vendor**	**Software /Infrastructure Provider**
	Functional properties		
Functions	the tasks of a software application needs	+	+
Dependencies	other components that the component depends on in order to function. A dependeny may exist within a computer system (e.g. applications depend on operating systems, and hardware). Dependencies may also exist between components of different computer systems (e.g. a web application on one server may need a database from another services.)		+
Conflicts	the set of components that a component cannot co-exist with in the same system.		+
	Non-functional properties		
Resource	the amount of resources needed for the component to function. For example, an MPI application may require a minimum numbers of processes in order to properly function. It can only also effective up to a maximum number of processes.		+
Cost	the amount of money, or equivalent credit, for the needed functions	+	+
Quality	the quality expected from or promised for the application and related services	+	+
Time	the period in which the application and related services can be used	+	+
Rights of use	the rights associated with that the applications and results	+	+

Table I
PROPERTIES SPECIFIED BY END USER/SERVICE VENDOR AND SOFTWARE/INFRASTRUCTURE PROVIDER

C. *Formal Models for Elastic Components and eHPA*

This section introduces a formal model of elastic components and eHPA. The current model only takes into account the following properties: functional capabilities, internal and external dependencies, conflicts, resources, cost, quality, available time and usage rights.

1) eComponent: Let ec be an elastic component, ec can be denoted as:

$$ec = \{F_p, E_p\}$$

where, F_p and E_p are functional and elastic property groups of the component ec, and:

$$F_p = \{F, D_i, D_x, C\}$$

where, F, D_i, D_x, and C are functions, internal dependencies, external dependencies and conflicts associated with the component ec respectively.

$$E_p = \{R_e, C_e, Q_e, T_e, Rt_e\}$$

where, R_e, C_e, Q_e, T_e, and Rt_e represent elastic resources, cost, quality, available time and usage rights, respectively.

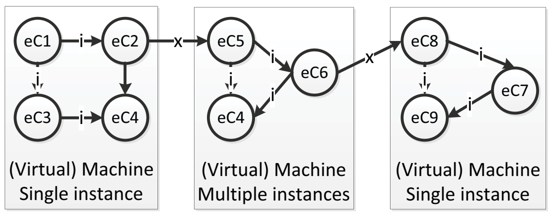


Figure 2. An illustrating example of eHPA with (i) internal and (x) external dependencies

2) eHPA: An *eHPA* is a collection of elastic components, linked by their dependencies and associated with a set of elastic properties. Formally, an *eHPA* can be represented as a graph, as illustrated in Figure 2. Each node of the graph is an elastic component. Each edge of the graph is a dependency, either internal or external. Interal dependencies are internal within a computer system. For example, an MPI application needs MPI runtime environment. Components linked by internal dependencies are tightly coupled, and reside together within a machine, as shown in the figure. External dependencies exist in form of data or control dependencies. Components linked by external dependencies are loosely coupled, and reside on different machines. As discussed, an eHPA can span on multiple computer systems. An eHPA also has its own set of elastic properties that are aggregated from the properties of its constituting components. Let ehpa be an eHPA, then:

$$ehpa = \{eC, D, E_p\}$$

where, eC, D and E_p is a set of constituting elastic components, dependencies and elastic properties of the eHPA, respectively. The graph derived from the set of components eC and the set of dependencies D. We denote a dependency d indicating that component ec_i depends on component ec_j as follows:

$$d = ec_i \rightarrow ec_j$$

We use $d_i = ec_i \xrightarrow{i} ec_j$ and $d_x = ec_i \xrightarrow{x} ec_j$ to indicate internal and external dependencies, respectively.

D. *Elastic Measurements and Aggregation*

1) Resource: For the sake of simplicity, resource is measured by the number of processes that a component needs in order to properly perform its functions. If the component can be elastic in terms of resources, R_e will be represented by a range: $R_e[m..n]$, where $0 < m < n$. If two different components, ec_1 and ec_2 have different resource elasticity, such as:

$$ec_1(R_e) = [m_1..n_1] \text{ and } ec_2(R_e) = [m_2..n_2]$$

then, depending on the types of dependencies that link the two components, the aggregated resource elasticity $ec(R_e)$ will be decided accordingly. If the two components are linked entirely by internal dependencies (they will be together on a single machine), the aggregated resource elasticity will be:

$$ec(R_e) = [max(m_1, m_2)..min(n1, n2)]$$

The aggregated result will not be valid if $max(m_1, m_2) > min(n_1, n_2)$. It means that the two components should not be aggregated. In practice, resources can be over or under provisioned, and the system can still work. However, in such a case, it cannot be guaranteed that their functions and quality will be performed as expected.

If the two components are linked by an external dependency, we calculate the aggregated resource elasticity as:

$$ec(R_e) = [(m_1 + m_2)..(n1 + n2)]$$

When an *eHPA* span on multiple systems, resource elasticity aggregation should be computed for components on each system first. Then, the aggregation between systems will be calculated later.

2) Cost: Elasticity of cost of a component is measured by a positive range of real numbers, and represented as $C_e[x..y]$, where $0 < x < y$. The aggregated cost of the two components ec_1 and ec_2 is calculated as below:

$$ec_1(C_e) = [x_1..y_1] \text{ and } ec_2(C_e) = [x_2..y_2]$$

then,

$$ec(C_e) = [(x_1 + x_2)..(y_1 + y_2)]$$

3) Quality: Quality can be understood as performance, reliability, security, quality of output, etc. For simplicity, enumerated positive integer values from 1 to 5 are used to describe the degree of quality a component can deliver. One (1) is for the lowest and five (5) is for the highest quality. The elasticity of quality is also represented by a range $Q_e[m..n]$, where $1 <= m <= n <= 5$. The aggregated quality of two components are calculated as:

$$ec(Qe) = [min(m_1, m_2)..min(n_1, n_2)]$$

4) Available Time: Available time is the period that the component is available for use. Elasticity of available time is measured by a positive range from zero, and represented as $T_e[t_1..t_2]$, where $0 <= t_1 <= t_2$. This expression means that the component is available for use in between time t_1 and t_2. Aggregated available time of two components $ec_{1T_e}[t_{11}..t_{12}]$ and $ec_{2T_e}[t_{21}..t_{22}]$ is calculated as:

$$ec(T_e) = [max(t_{11}, t_{21})..min(t_{12}, t_{22})]$$

The aggregated available time will not be valid if $max(t_{11}, t_{21}) > min(t_{12}, t_{22})$.

5) Right: Rights associated with a component (e.g. specified as license terms) are a set of terms, represented as $ec(Rt_e)$. Aggregated rights of the two components in the intersection of the two sets.

$$ec(Rt_e) = ec_1(Rt_e) \cap ec_2(Rt_e)$$

The aggregated value is invalid if the two sets of rights do not overlap each other.

IV. Prototype Implementation

A. Elastic Component Information Model

To realize the conceptual model described in Section III, we extend the ontology used in [11]. The ontology is extended with object and data properties to model elastic properties of components. In order to populate instance information for the ontology, we gather information from different cloud providers.

B. eHPA Composition Tool

We have developed a Composition Tool for composing *eHPAs*, which accepts user requests consisting of functional information, resource, cost, quality, available time and rights. The core of this tool is a Composer, which consists of different plug-able modules for different processing purposes. Currently, four different modules have been implemented, including Dependency and Conflict Solver, Candidate *eHPA* Assembler, Elasticity Calculator and Elasticity Filter for resolving dependencies, analyzing elastic requirements of the users and properties of *eHPAs*.

C. Composition Algorithms

Let UR represent a user request, then, $UR = \{P, E_p\}$ where P is the set of requested partition, and E_p is the set of elasticity requirements. Each requested partition $p_i \in P$ contains a set of initial requested components eC. Let $G\{eC, D\}$ be the dependency graph extracted from the ontology, where each node $ec_i \in eC$ is a component and each edge $d_i \in D$ is a dependency that exists between two component. The graph G may contains disconnected subgraphs. The following funtions are defined:

- *resolveDependencies(ec)*: resolves dependencies of component ec and returns a set of components that ec depends.
- *checkConflict(Set<eC>, ec)*: checks for conflict between component ec and a set of existing components eC; returns true if there is a conflict, false otherwise.
- *findCandidatePartition(Set<eC>)*: finds candidate partitions that contains the initial set of components eC; returns a set of candidate partitions.
- *formCandidateeHPA(Set<Set<P>>)*: forms candidate *eHPAs* from candidate set of partitions P; returns a set of candidate *eHPAs*.
- *checkERequirements(Set<eHPA>,Ep)*: checks for elasticity requirements on candidate set of *eHPAs* against

the set of elastic properties Ep; returns a set of *eHPAs* that satisfy the requirements.

Algorithm 1 eHPA composition algorithm

```
1: Initialize set of partition collection $pCol$
2: for all $rp_i \in UR(P)$ do
3:     Initialize set of candidate partition $cPart$
4:     $cPart$ = findCandidatePartition($rp_i(eC)$)
5:     Add $cPart$ to $pCol$
6: end for
7: Initialize set of candidate eHPA $ceHPA$
8: $ceHPA$ = formCandidateeHPA($pCol$)
9: Initialize set of qualified eHPA $qeHPA$
10: $qeHPA$ = checkERequirements($ceHPA$, $UR(E_p)$)
11: return $qeHPA$ as the result of assembling process
```

The overall composition algorithm is designed as shown in Algorithm 1. The most significant part of Algorithm 1 is the function *findCandidatePartition*. This function traverses the dependency graph $G\{eC, D\}$ to looks for all possible partitions from an initial set of requested components. The process used by this function is described in Algorithm 2.

Algorithm 2 Find candidate partition from a given set of requested components eC

```
1: Initialize set of candidate partition $cP$
2: while The graph $G\{eC, D\}$ is not exhaustively searched do
3:     Initialize set of temporarily components $tempComp$
4:     for all $ec_i \in eC$ do
5:         $tempdep$ = resolveDependencies($ec_i$)
6:         for all $ec_j \in tempdep$ do
7:             if Not checkConflict($tempComp$, $ec_j$) then
8:                 Add $ec_j$ to $tempComp$
9:             else
10:                Clear $tempComp$
11:                Break for loop
12:            end if
13:        end for
14:    end for
15:    if $tempComp$ is not empty then
16:        Form a candidate partition $p$ from $tempComp$
17:        Add $p$ to $cP$
18:    end if
19: end while
20: Return the set of candidate partition $cP$
```

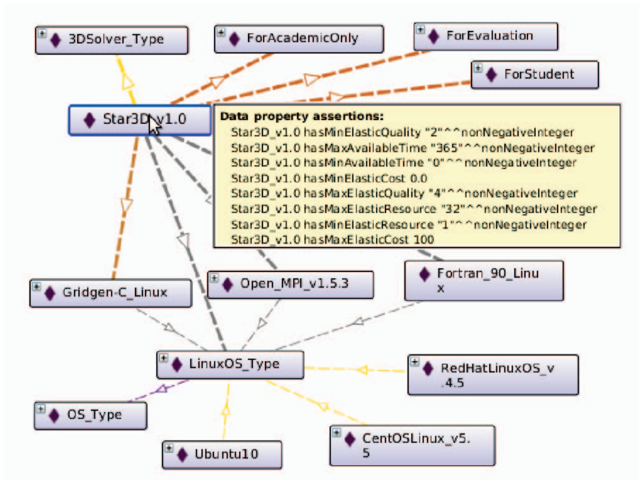
V. Experiments

A. Modeling Star3D

In this section, we illustrate an modeling example for Star3D program, which is developed for solving Euler equations in the cases of 3D flows. It is an MPI parallel program, written in Fortran. To use with Star3D, a CAD model needs to be pre-processed to generate a mesh for computation. Many different tools can be used for generating meshes, ranging from commercial licensed to free and open source software. The data generated by Star3D is then post-processed by analysis and/or visualization tools. Similar to pre-processing, there are many different tools can be used for post-processing.

1) Modeling Dependencies: Star3D component can only function properly if the system that it is installed and configured has a compatible Fortran 90 compiler, MPI runtime. These dependencies are modeled as internal dependencies, as they are mandatory and reside internally within the same system as Star3D. Star3D application takes output from Gridgen-C as its input. Its output is then further processed by ParaView for visualization. To complete an experimental process with Star3D, we can say that Star3D depends on Gridgen-C and ParaView. However, these dependencies are data/control based, and not mandatory for Star3D. Therefore, these dependencies are modeled as external dependencies. Other components such as Open MPI, ParaView, Gridgen-C or Fortran 90 also have their own set of dependencies. Each set of dependencies is a directed and connected graph. Linking many sets of dependencies of different components will result in a big dependency graph. Using this dependency graph, we can identify necessary components for running an application.

Dependency relationship can also exist between a component and an abstract type, which is created to represent components of with same functionality. For example, *LinuxOS Type* can be created to represent all Linux family operating systems. As Star3D can be used with any operating system of type 'Linux OS', we can model that Star3D version 1.0 is dependent on *LinuxOS Type*, as in Figure 3.



Data property assertions:

```
Star3D_v1.0 hasMinElasticQuality "2"^^nonNegativeInteger
Star3D_v1.0 hasMaxAvailableTime "365"^^nonNegativeInteger
Star3D_v1.0 hasMinAvailableTime "0"^^nonNegativeInteger
Star3D_v1.0 hasMinElasticCost 0.0
Star3D_v1.0 hasMaxElasticQuality "4"^^nonNegativeInteger
Star3D_v1.0 hasMaxElasticResource "32"^^nonNegativeInteger
Star3D_v1.0 hasMinElasticResource "1"^^nonNegativeInteger
Star3D_v1.0 hasMaxElasticCost 100
```

Figure 3. Visualizing Star3D and its relationships using OntoGraf

2) Modeling Elastic Properties: The current version of Star3D is programmed to work in parallel up to 32 processes.

It is also able to work with single process as a sequential program. Hence, in resource dimension, the current version of Star3D can scale from one to 32 processes. The authors develop the software for their own use and are also willing to share its current version with others in their community for academic purpose, evaluation and student usage free of charge, for an unlimited time. Therefore, in terms of usage right, the users can use the software for these two purposes. The cost of the software is free, and the available time is unlimited. Quality of a software component is reflected in a number features such as performance, correctness, reliability, support, etc. In the current model, we use discrete integer values from 1 to 5 to represent the degree of quality associated the software, from the lowest to the highest. The current version of Star3D, we subjectively rank it from 2 to 4, inclusively.

B. *Star3D-based eHPAs*

Let us examine the effect of using elasticity properties in the composition process. Supposed that we want to run Star3D on Amazon EC2 cloud with Amazon Linux machine image, and that Gridgen-C application is chosen as a tool for mesh generation to produce input for Star3D. Gridgen-C will run on a separate system from Star3D-based *eHPA*. To support the composition process, we have created 75 components in the knowledge base, using information from the Internet.

There are 12 different combinations of available components that satisfy the request. Elastic properties of resulted *eHPAs* are shown in Table II. Four groups of solutions have similar values: {2,5},{8,11}, {0,3,6,9} and {1,4,7,10}. Although all solutions of a group have the same elastic properties, the sets of components that make up the solutions are different. The most noticeable difference between these groups is in cost. The cost of Star3D program is free for academic, student and evaluation use. However, to run Star3D, we also need an MPI runtime, Fortran compiler and a Amazon cloud platform. In our experiment, the cost of running Star3D includes basically the cost of using Amazon EC2 cloud (10 US cent/core/hour, allocation size varied from 20 to 1000 cores), and the cost of Portland Fortran compilers (if selected). Solution 2, 5, 8 and 11 use a free version of Fortran, as shown in Figure 4. Therefore, the cost is only made up by using Amazon EC2. Other solutions use different parallel versions of Portland Fortran (for 64 and 256 processes), as in Figure 5. Therefore, the costs are varied. Other elastic properties such as resource, quality, available time, and right are constrained by properties of Star3D and Gridgen-C. We use the value of 365 to denote unlimited available in one year time frame.

C. *Discussions*

In this paper, we address the issues of elasticity of applications when running in cloud computing environments. Five different properties: resource, cost, quality, right and time are proved to be elastic, and should be addressed during the preparation for deploying and running application on clouds. In this paper, for demonstration purpose, we model these attributes in their most simplest forms. In reality, each elastic property can be further modeled into sub-dimensions. For example, instead of using single values for max and min cost, cost can be further analyzed into detail, such as using cost units (per core, per hour, per license, etc.). In addition, these elastic properties are not totally independent from each other. Cost is influenced by license type, numbers of current processes used, available time, etc. During the composition process, values of elastic properties of different components are aggregated for the resulted *eHPA*, and then re-applied back to all related components. The change of one elastic property (e.g. license) will influence other properties (e.g. cost). Although we are aware of these issues, they are not addressed in the current version of the prototype.

Apart from automatic finding different combination of software components for packing and running in clouds, the framework can also be used as an application modeling tools for cloud. By visualizing different required components of an application and their inter-relationships, the users can realize which components can be benefit from clouds and which one should be run on their own private platforms. They can then further customize the inputs to the composition processes to find better solutions.

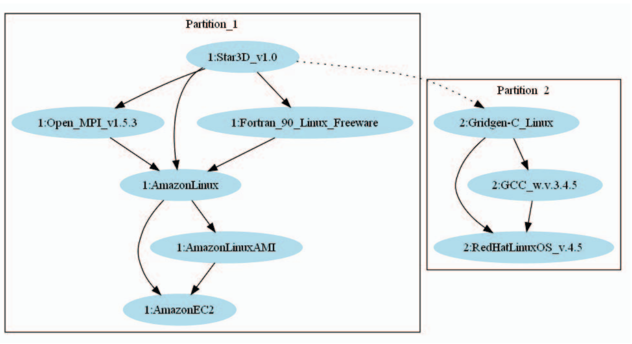


Figure 4. An *eHPA* solution using free a Fortran compiler (solution 8)

VI. Conclusions and Future Work

Supporting scientists to move and design their complex high performance applications for multiple cloud environments requires us not only to deal with complex software dependencies and conflicts but also to determine and characterize elastic properties of these applications. In this paper, we introduce techniques to build *eHPA* (elastic high performance applications) and to characterize them with multi-elasticity dimensions, covering resource, quality, cost, available time and usage rights. By utilizing these elastic properties, we can determine to which extend an *eHPA* in multiple clouds is elastic. Based on that, we can decide not

Solution	Cost		Resource		Quality		Right	Time	
No	Min	Max	Min	Max	Min	Max	Rights	Min	Max
2	2	100	21	33	2	4	ForAcademicOnly;	0	365
5	2	100	21	33	2	4	ForAcademicOnly;	0	365
8	2	100	21	33	2	4	ForAcademicOnly;ForEvaluation;ForStudent;	0	365
11	2	100	21	33	2	4	ForAcademicOnly;ForEvaluation;ForStudent;	0	365
0	2901	10249	21	33	2	4	ForAcademicOnly;	0	365
3	2901	10249	21	33	2	4	ForAcademicOnly;	0	365
6	2901	10249	21	33	2	4	ForAcademicOnly;	0	365
9	2901	10249	21	33	2	4	ForAcademicOnly;	0	365
1	4701	8899	21	33	2	4	ForAcademicOnly;	0	365
4	4701	8899	21	33	2	4	ForAcademicOnly;	0	365
7	4701	8899	21	33	2	4	ForAcademicOnly;	0	365
10	4701	8899	21	33	2	4	ForAcademicOnly;	0	365

Table II
EXPERIMENT RESULTS

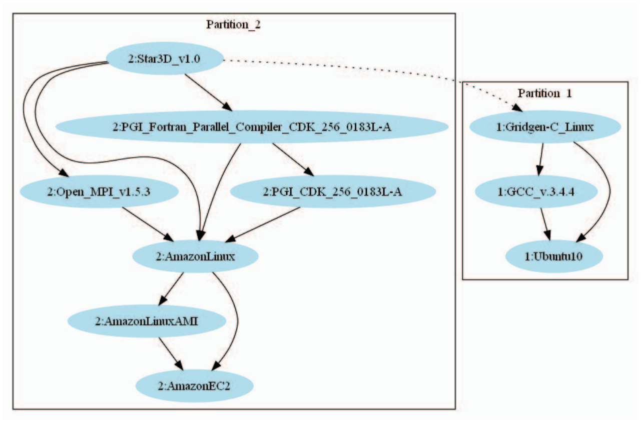


Figure 5. An eHPA solution using Portland Fortran compiler, licensed for use up to 256 MPI processes (solution 4).

only which software components should be used but also when to use which cloud environments.

Our future work focuses on enriching our information model to support composition and modeling. Furthermore, we are currently developing and integrating our techniques to runtime packing and deploying *eHPAs*.

REFERENCES

[1] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim, "Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows," in *AAAI*. AAAI Press, 2007, pp. 1767–1774.

[2] S. Bowers, B. Ludascher, A. H. H. Ngu, and T. Critchlow, "Enabling scientificworkflow reuse through structured composition of dataflow and control-flow," in *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ser. ICDEW '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 70–. [Online]. Available: http://dx.doi.org/10.1109/ICDEW.2006.55

[3] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren, "The cca component model for high-performance scientific computing," *Concurr. Comput. : Pract. Exper.*, vol. 18, pp. 215–229, February 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1107430.1107433

[4] Y. Simmhan, C. van Ingen, G. Subramanian, and J. Li, "Bridging the gap between desktop and the cloud for escience applications," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 474–481. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.72

[5] J. D. McGregor, L. M. Northrop, S. Jarrad, and K. Pohl, "Guest editors' introduction: Initiating software product lines," *IEEE Software*, vol. 19, no. 4, pp. 24–27, 2002.

[6] J. Rao and X. Su, "A survey of automated web service composition methods," in *Semantic Web Services and Web Process Composition*, ser. Lecture Notes in Computer Science, J. Cardoso and A. Sheth, Eds. Springer Berlin / Heidelberg, 2005, vol. 3387, pp. 43–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30581-1_5

[7] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer, "Supporting runtime system adaptation through product line engineering and plug-in techniques," in *ICCBSS*. IEEE Computer Society, 2008, pp. 21–30.

[8] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A generic configurable customizable composite cloud application framework," in *OTM Conferences (1)*, ser. Lecture Notes in Computer Science, R. Meersman, T. S. Dillon, and P. Herrero, Eds., vol. 5870. Springer, 2009, pp. 357–364.

[9] C. Bunch, N. Chohan, C. Krintz, and K. Shams, "Neptune: a domain specific language for deploying hpc software on cloud platforms," in *Proceedings of the 2nd international workshop on Scientific cloud computing*, ser. ScienceCloud '11. New York, NY, USA: ACM, 2011, pp. 59–68. [Online]. Available: http://doi.acm.org/10.1145/1996109.1996120

[10] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," in *ISPAN*. IEEE Computer Society, 2009, pp. 4–16.

[11] T. V. Pham, H. Jamjoom, K. Jordan, and Z.-Y. Shae, "A service composition framework for market-oriented high performance computing cloud," in *ACM HPDC 2010*, Chicago, USA, June 2010.