SOCA (2017) 11:87–100
DOI 10.1007/s11761-016-0199-0

ORIGINAL RESEARCH PAPER

MADONA: a method for automated provisioning of cloud-based component-oriented business applications

Hind Benfenatki[1] · Catarina Ferreira Da Silva[1] · Gavin Kemp[1] · Aïcha-Nabila Benharkat[2] · Parisa Ghodous[1] · Zakaria Maamar[3]

Received: 1 November 2015 / Revised: 25 August 2016 / Accepted: 30 August 2016 / Published online: 13 September 2016

Abstract Service-oriented computing and cloud computing offer many opportunities for developing and deploying applications. In this paper, we propose and describe a component-oriented method for automated provisioning of cloud business applications. The method covers the whole application's lifecycle and is based on cloud orchestration tools that manage the deployment and dependencies of supplied components. We aim to reduce the necessary technical knowledge for provisioning component-oriented cloud applications. To this end, we extend Linked Unified Service Description Language to describe services for matching user's requirements. We adopt a real case study to show the feasibility of the method.

Keywords Cloud computing · Linked services · Component-oriented business applications development · Service description · Cloud orchestration tool

✉ Hind Benfenatki
hind.benfenatki@liris.cnrs.fr

Catarina Ferreira Da Silva
catarina.ferreira-da-silva@liris.cnrs.fr

Gavin Kemp
gavin.kemp@liris.cnrs.fr

Aïcha-Nabila Benharkat
nabila.benharkat@insa-lyon.fr

Parisa Ghodous
parisa.ghodous@liris.cnrs.fr

Zakaria Maamar
zakaria.Maamar@zu.ac.ae

[1] Univ Lyon, Université Claude Bernard Lyon 1, LIRIS UMR 5205 CNRS, 69621 Villeurbanne Cedex, France

[2] LIRIS, CNRS, UMR5205, INSA - Lyon, 69621 Lyon, France

[3] Zayed University, Dubai, UAE

1 Introduction

Today's business applications are typically complex calling for the collaboration of several independent components, providing each a separate functionality. Service-oriented computing (SOC) refers to these components as services that can be assembled in a loosely coupled way. In conjunction with using SOC to address integration problems, cloud computing has emerged as another way of helping enterprises access hardware and software resources on demand and pay-per-use. There is a consensus in the R&D community that both SOC and cloud computing constitute a successful combination for the management of Service-Oriented Cloud Computing Architecture (SOCCA) [1]. On the one hand, SOC automates the development of composite applications. On the other hand, cloud computing provisions deployment environments for these applications.

Several efforts are put into developing support tools and languages for the deployment of applications on cloud environments such as Amazon Web Services (AWS) CloudFormation [2], Heat [3], TOSCA [4], and Juju [5]. These tools and languages use scripts to describe components of the future cloud applications and necessary infrastructure resources. Unfortunately, developing such scripts requires a good technical knowledge of both the deployment language and the necessary components of the applications. It is worth noting that these scripts only allow a static composition of the underlying application components. As a result, they do not accommodate changes in components smoothly. This constitutes a major limitation to the use of scripts in dynamic environments. In fact, an implementation change or upgrade of a component can have an impact for instance, on its composability, necessary resources for its deployment, and its configurability. These changes cannot be taken into account

automatically if we would work with a ready to use script only.

In this paper, we leverage the benefits of cloud computing and Service-Oriented Architecture (SOA) in order to reduce the necessary technical knowledge and human involvement in provisioning component-oriented cloud applications (component composition and application deployment). Our first contribution is MADONA that stands for Method for AutomateD prOvisioning of cloud-based component-oriented busiNess Applications. MADONA consists of the following phases: (a) Requirement elicitation; (b) Application components discovery; (c) Integration of new components if necessary; (d) Composition plans generation; (e) Infrastructure as a Service (IaaS) discovery for application hosting; (f) Composition plans ranking and selection; (g) Application configuration and customization; (h) Automatic deployment of cloud application; and (i) Test and Validation of the deployed application. Human involvement and technical knowledge are reduced since MADONA phases are automated. In fact, the user intervenes only in requirement elicitation phase and when the application is deployed and ready to use.

The application components discovery and IaaS discovery phases rely on Linked Unified Service Description Language (USDL) [6–8] descriptions of available application components and IaaS. According to Thoma et al. [9] *Linked USDL is the only current standardization effort driven by large corporations (such as SAP, Siemens and Attensity) with the goal of expressing not only purely functional aspects of a service, but also the business and operational aspects. A comprehensive introduction into each can be found in* [10]. Our second contribution aims to increase composability of a component [11] to favor the development of cloud component business applications. We extend Linked USDL to track relations that a component *can* have and *must* have with other peers. Based on this extension, our third contribution is an algorithm for generation of components composition plans meeting user's requirements and taking into account each component's composition constraints and possibilities.

The rest of this paper is organized as follows. Section 2 describes existing service description languages and application development and deployment approaches. Section 3 presents MADONA's phases. Section 4 describes the implementation of MADONA. Sections 5 and 6, respectively, evaluates and discusses our work. Section 7 draws final conclusions.

2 Related work

Our literature review resulted into classifying service-oriented cloud application provisioning into two categories: (1) cloud application development environments and architectures [12, 13] and (2) cloud application deployment languages and tools [4,5].

Several efforts are put into providing environments for developing service-oriented cloud applications. SOCCA [1] combines SOA and cloud computing so that clouds can interoperate with each other when developing service-oriented applications. In [1], Software as a Service (SaaS) applications are built by assembling services which, unlike traditional SOA, are packages that can be deployed on different clouds. We refer to these services as components. A cloud broker is used for discovering the necessary cloud platform and infrastructure resources for SaaS components. In [13], Sun et al. describe Service-Oriented Software Development Cloud (SOSDC); it is a cloud platform for developing a service-oriented software and dynamic hosting environment. SOSDC's architecture encompasses the three levels of cloud services. IaaS level provides infrastructure resources. Platform as a Service (PaaS) level provides App Engine for hosting, testing, running, and monitoring service-oriented software applications. And SaaS level provides online service-oriented software development environment. Once an application is built, the developer may request an App Engine for hosting the application by specifying requirements like Virtual Machine (VM) images and software appliance. Zhou et al. [14] extend the conventional architecture of cloud computing by inserting "Composition as a Service" (CaaS) layer between SaaS and PaaS layers for dynamic composition of services. The CaaS layer provides users with a Cloud-based Middleware for Dynamic Service Composition (CM4SC) that allows automatic service discovery and automatic and dynamic composition of Web services.

Other works focus on the deployment of cloud applications. AWS CloudFormation [2] and Heat [3] (OpenStack [15] module) describe the necessary infrastructure for supporting cloud application execution. In [4], Binz et al. describe the Topology and Orchestration Specification for Cloud Applications (TOSCA) OASIS standard language [16, 17]. TOSCA defines a topology for deploying cloud applications in terms of components and relations between these components. TOSCA, also, allows to describe management plans, which can be executed automatically to deploy, configure, and operate a cloud application. Juju [5], an open source orchestration management tool allows to deploy, configure, and compose software components on the cloud using high-level scripts (close to natural language, e.g., "juju deploy mysql"). These scripts call charms, which describe YAML Ain't Markup Language (YAML) configuration files and hooks. Juju environment can be bootstrapped on many clouds: Amazon Elastic Compute Cloud (EC2) [18], HP Cloud Services [19], Microsoft Windows Azure [20], OpenStack, etc.

The aforementioned works are used to deploy cloud applications but require a good prior knowledge of (1) the

application components [2–5], (2) the necessary environment for deploying each component [4], and (3) the infrastructure resources for component deployment [2,3].

Our work aims to reduce the necessary technical knowledge for provisioning component-oriented cloud applications. We rely on Linked USDL to describe application components and IaaS. Linked USDL describes business and cloud services and is based on Linked Data principles [21], which eases its extension. It reuses several linked vocabularies to describe business (e.g., legal issue and provider information), operational (e.g., service features and operations), and technical (e.g., used ports and protocols) aspects. Furthermore, Linked USDL is based on HTTP URIs allowing a global service identification, and on HTTP URLs and RDF to access service descriptions in a global, standard, and uniform manner [22]. However, Linked USDL does not describe composition interactions that a component has with peers. Nguyen et al. [23] describe services using blueprints. A blueprint describes a service's offers, requirements, and performance constraints. Nonetheless, Nguyen et al. cover environment and composition constraints in the requirement description. In fact, requirements can be either database (composition) or Web server (environment). Encompassing environment and composition constraints in the same concept is not suitable while automating the composition of components.

3 Method for automated provisioning of component-oriented cloud business applications

This section describes first services and then MADONA in terms of main phases, rationales, and illustrations.

3.1 Overview

Figure 1 illustrates MADONA's phases. MADONA is built upon SOA and cloud computing principles so that automatic provisioning of component-oriented cloud applications is achieved (i.e., component composition and deployment). It takes user's requirements as input and generates composition plans as output for future composite applications to deploy on the cloud. A composition plan is an abstract application that assigns components to IaaS. If there are not components that meet user's functional requirements, assistance is provided to the user so that she integrates new components into the service repository. Juju charms store [24] is an example of service repository that offers open source application components. We enrich this store by adding cloud services such as IaaS for hosting the generated application. We refer to the set of application components and IaaS as services. We use cloud orchestration tools (e.g., Juju) to deploy an abstract application on the preselected IaaS. Indeed, orchestration tools allow software deployment, integration, and scaling on

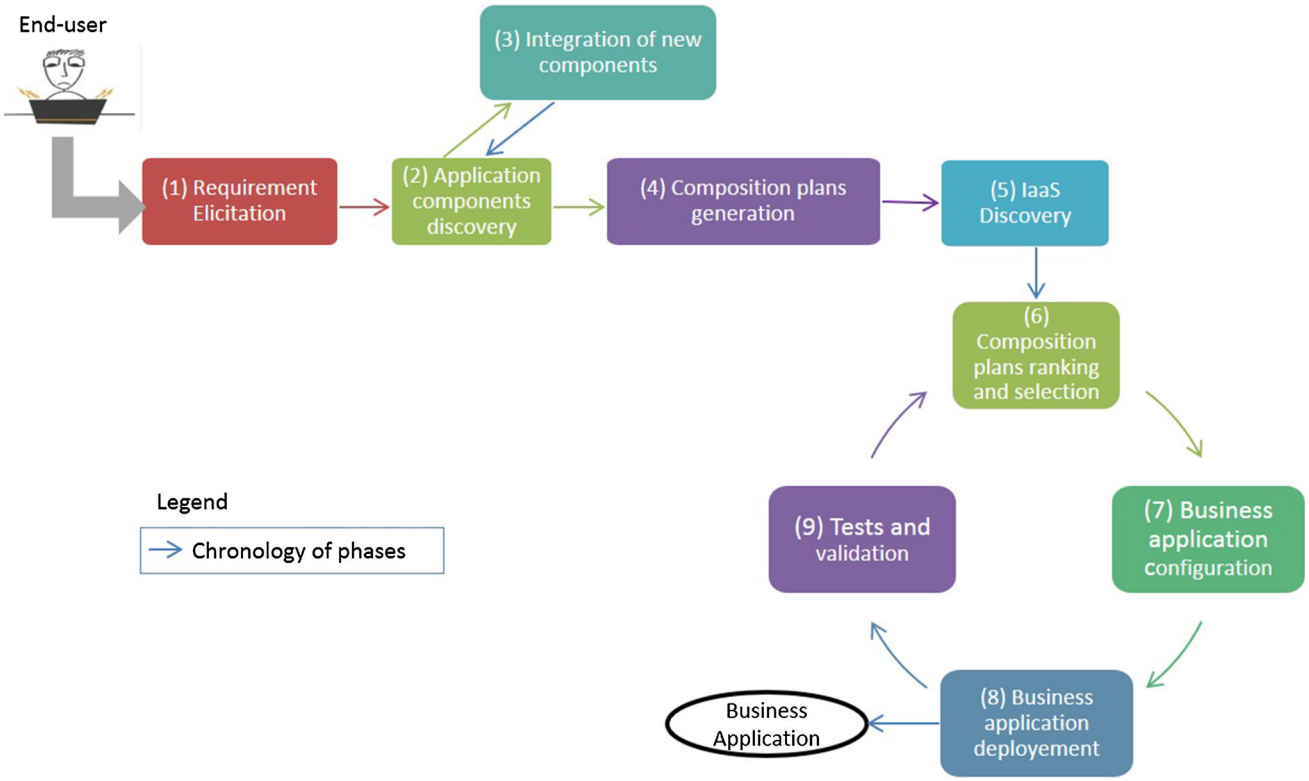

Fig. 1 MADONA's main phases

several clouds. Deployment constraints represent IaaS upon which an orchestration tool can deploy application components.

Our case study refers to a user who needs to provision a project management system for a software development company. The user, also, needs a version control system so that developers can store, retrieve, and merge different versions of application development. A particular project management system may require components, which in turn may require other components to function. It can be composed with a particular version control system (VCS) or may not have any possibilities to be composed with a VCS. We do not expect from the user to specify these technical details of this requirement. Instead, the user focuses on high-level functional (such as project management and version control systems) and nonfunctional requirements (e.g., application cost). The user prefers to host the generated application in Europe, on "Amazon" [18] due to successful previous uses, and to pay application use costs in "Euro," below 50 euros per month. Three challenges are associated with this scenario: (1) how to provision a project management application dynamically on the cloud with minimal human intervention, (2) how to specify and model the project management scenario's functional and nonfunctional requirements, and (3) how to adapt or extend service description languages to facilitate the selection and composition of the project management's components.

3.2 Description of services

Providers are expected to describe supplied application components and IaaS. Each component has deployment and configuration scripts and an additional script that connects it to other components. We extend USDL core module [25] (Fig. 2) to track the composition interactions that a component has with peers. We consider two types of interactions when describing a component: constraints and possibilities. Constraints refer to an application's necessary components, environment, and resources. And possibilities refer to optional components that can be composed together. We consider that environment constraints (e.g., Web server) are automatically integrated into the deployment scripts. Composition constraints concern the application components.

To automate the configuration and deployment of components, we describe configurable parameters and minimal required resources for each component. In Fig. 2, the new properties are numbered from 1 to 7 and the new concepts are at the top of these properties. Each component is described as follows S = {CC, CP, CCP, MRR}.

– CC is the set of Composition Constraints on a component. Constraints are either hard or soft. The former describe the components that must be put together, e.g., MediaWiki and MySQL database. The latter describe choices like MSSQL versus PostgreSQL versus MySQL databases to connect to Joomla.
– CP represents Composition Possibilities of a component, i.e., compositions that a component can have with peers (e.g., memcached, a memory caching component represents a possible composition of MediaWiki).
– CCP describes Component Configurable Parameters, like name and logo. Each configurable parameter is described with a name and type of the Web page element (e.g., text area or drop-down list) which serves to generate configuration Web interfaces (Sect. 3.3.7).
– MRR represents the Minimal Resource Requirements of a component (e.g., number of CPU and memory size).

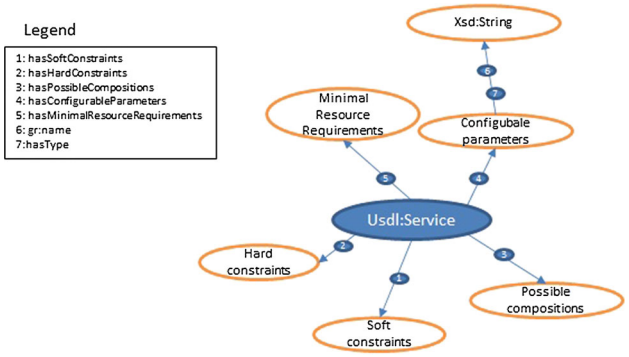

Fig. 2 Extended linked USDL core module

```
<http://mydomain.fr/usdls/projectman>
<ProjectMan>
    a       usdl:service ;

    gr:name "ProjectMan" ;

    usdl:hasDescription "ProjectMan is a project
    management engine." ;

    usdl:hasHardConstraint
    [
       a usdl:service;
       gr:name "MySQL" ;
    ]

    usdl:hasHardConstraint
    [
       a usdl:service;
       gr:name "MyCRM" ;
    ]

    usdl:hasPossibleComposition
    [
       a usdl:service;
       gr:name "MyVCS" ;
    ]

    usdl:hasPossibleComposition
    [
       a usdl:service;
       gr:name "Memcached" ;
    ]

    usdl:hasMinimalResourceRequirements
    [
       a usdl:MinimalResourceRequirements
       usdl:hasCpu "4";
       usdl:hasMemory "1G";
    ]

    hasConfigurableParameter
    [
       a usdl:ConfigurableParameter
       gr:name "name"
       usdl:hasType "text area"
    ].
```

Listing 1 ProjectMan description via .usdl file

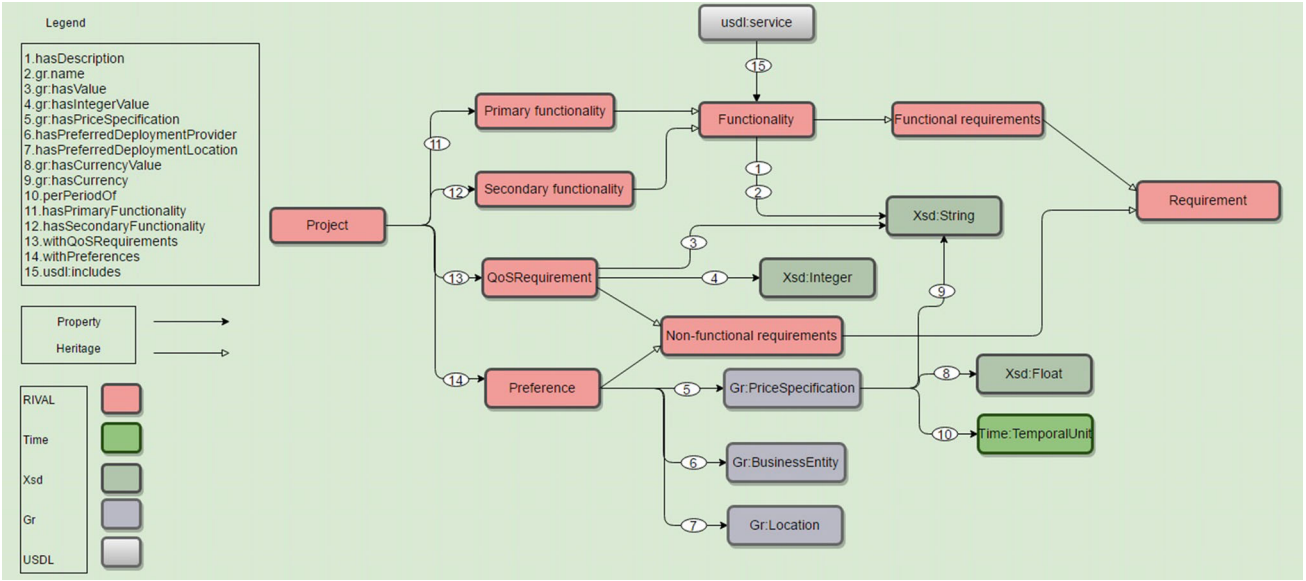

Fig. 3 RIVAL's concepts and properties

Services' descriptions are stored in .usdl files. Listing. 1 illustrates the description of ProjectMan, a project management component, using the extended Linked USDL. ProjectMan has two possible compositions with, respectively, MyVCS (lines 22–26), a version control component and Memcached (lines 28–32), a memory caching component. It requires MySQL database to store data (lines 10–14) and MyCRM (lines 16–20), a CRM component, to link employees with the customer who initiated the project. ProjectMan component requires minimal VM configuration with 4 CPU and 1 Go memory (lines 34–39) and can be personalized by its name (lines 41–46). In addition, MyCRM needs to communicate with EmployMan (an employees' management system) to identify the employees involved in a project. MyCRM, EmployMan, and MyVCS, each requires MySQL database.

3.3 MADONA's phases

This subsection illustrates MADONA's phases (Fig. 1) from requirement elicitation to tests and validation of deployed application.

3.3.1 Requirement elicitation phase

This phase describes user's requirements for the future cloud applications using RequIrement VocAbuLary (RIVAL). RIVAL formalizes these requirements using linked vocabularies and introduces a distinction between primary and secondary functionalities. These latter help, respectively, select primary components and their possible compositions. The functionality of the desired application is considered as primary, e.g., a project management functionality in our scenario. Any additional functionality to this project management application is considered as secondary, e.g., requiring a version control system with the project management component to store several development versions of a project. Only one primary functionality is allowed by project. Several secondary functionalities can be associated with it. Fig. 3 illustrates RIVAL classes that describe vocabulary's concepts and properties describing relations between concepts.

It is challenging for a user to estimate acceptable tolerance thresholds for QoS parameters such as service availability and data integrity. In fact, users always aim for a maximum quality. For these reasons, we use weights that a user affects to QoS parameters according to her priority in the way that the sum of affected weights equals 10. The choice of the amount of weights of QoS parameters to equal to 10 is due to the simplicity, in our view, to distribute 10 points rather than a percentage.

User preferences concern application cost and deployment. The former concerns the maximum cost authorized, currency, and purchase period. The latter concerns the preferred provider and location.

Requirements are translated into a .rival file. Listing. 2 illustrates a .rival file generated for the project management scenario. Lines 3 and 4 describe, respectively, the user's primary and secondary functionalities. The user's preferences are described from lines 6 to 17, including the preferred deployment provider and location (lines 8–9), and price specification (lines 10–16). Lines 19–39 describe the user's QoS requirements. The .rival file is used in the application component discovery phase.

User's requirements may generate conflicts among themselves. A conflict occurs when requirements generate incompatibilities between common software attributes [26] or when

performing an activity that prevents the execution of another one [27]. It can be due to inconsistency in the specifications in case of multiple stakeholders [28]. In our work:

- QoS requirements are expressed as weights rather than as precise values for each QoS parameter in order to try to avoid conflicts, such as imposing multiple authentications (security) and requiring a minimum time to login at the same time.
- If conflicts exist between two components, they will not be reflected on the generated application. In fact, the generation of composition plans consists of composing components that *can* be composed and the ones that *have to* be composed (composition constraints and possibilities).

```
<http://mydomain.fr/rival/project_management>
<Project management scenario> a rival:Project
 rival:hasPrimaryFunctionality "project management";
 rival:hasSecondaryFunctionality "version control system";

 rival:withPreferences
 [   a rival:Preference;
     rival:hasPreferredDeploymentProvider "Amazon";
     rival:hasPreferredDeploymentLocation "Europe";
     gr:hasPriceSpecification;
     [
        a gr:PriceSpecification;
        gr:hasCurrency "euro";
        gr:hasCurrencyValue "50";
        usdl:perPeriodOf "month";
     ]
 ]

 rival:withQoSRequirements
 [
    a rival:QoSRequirement;
    gr:hasValue "Data_Privacy";
    gr:hasIntegerValue  "4"
 ],
 [
    a rival:QoSRequirement;
    gr:hasValue "Response_Time";
    gr:hasIntegerValue  "1"
 ],
 [
    a rival:QoSRequirement;
    gr:hasValue "Data_loss";
    gr:hasIntegerValue  "4"
 ],
 [
    a rival:QoSRequirement;
    gr:hasValue "Availability";
    gr:hasIntegerValue  "1"
 ].
```

Listing 2 Project management scenario described via .rival file

3.3.2 Application components discovery phase

This phase consists of looking for application components that meet user requirements' functionalities (primary and secondary). This requires matching the user requirements with existing components. For the sake of simplicity, we adopt a syntactic matching. Semantic matching is part of our future work and could be based on some well-defined techniques [29,30].

Listing. 3 illustrates the SPARQL Protocol and RDF Query Language (SPARQL) query that returns the components satisfying the user's requirements. The query construction is automatically done by the system. It follows these steps: check if the user has a preferred component meeting her requirements based on previous experiences. If so, the selection is done following the component's name rather than using keywords (lines 2–3); the discovery query also retrieves the composition constraints of components meeting the primary functionality (line 4); for each desired secondary functionality, selects the components meeting the latter and that can be composed with the primary components (lines 5–7).

```
1  SELECT  ?i ?e ?f WHERE {
2  ?x usdl:hasDescription "project management".
3  ?x gr:name ?e.
4  ?x usdl:hashardConstraints ?f.
5  ?x usdl:hasPossibleComposition ?g.
6  ?g usdl:hasDescription "version control system".
7  ?g gr:name ?i.}
```

Listing 3 SPARQL query for a project management application

The components discovery phase returns three key lists: (1) Matched Primary Components (MPC), (2) Matched Primary Components Constraints (MPCC), and (3) Matched Secondary Components (MSC): for each desired secondary functionality, an MSC is created.

While generating composition plans, we look for composition constraints per component involved in a composition plan via another SPARQL query (Sect. 3.3.4).

3.3.3 Integration of new components phase

The user can upload new components to the service repository when the existing ones do not meet her requirements. This phase consists of two steps.

1. Integration into the service repository of deployment, configuration, and composition scripts: these scripts are used, respectively, to automate the deployment and customization of a component, and the management of dependencies with other components. These scripts are developed by a component's provider and are uploaded to the service repository via a Web interface. This step is dependent on an orchestration tool. It allows, in our implementation, to integrate charms of new components into Juju store.
2. Component description: this is introduced via a Web form and automatically translated into .usdl files that constitute our repository of services.

3.3.4 Composition plans generation phase

We generate composition plans that meet user's requirements and components' constraints (Listing 4). A composition plan consists of functional and deployment parts. The former consists of a list of relations that connects components together. Composition plans' functional part is generated as follows: let i be the index of a primary component in MPC, n the number of desired secondary functionalities, and j the index of needed secondary functionalities. We associate with each primary component in MPC a composition plan. We generate

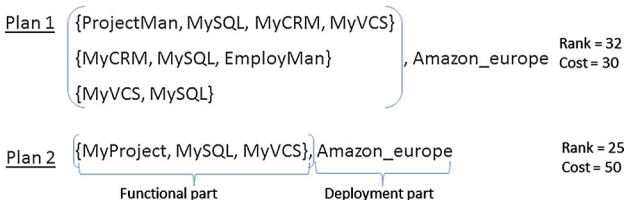

Fig. 4 Possible composition plans generated for the project management scenario

the possible combinations between MSC[i, j] ($j = 0$, $j < n$) so that one component from each MSC[i, j] is present. For each possible combination, we combine MPC[i] and MPCC[i] to generate the first relation of the plan. For each component (from the second component) of the first relation, we look for its composition constraints through a new SPARQL query. If a query's result is not null, a new relation is created containing the component and its constraints. The added relations are also checked for their composition constraints. The deployment part of a composition plan represents the IaaS upon which it can be deployed.

Figure 4 illustrates the composition plans generated for the running scenario. The functional part of the first plan is composed of three relations. The first one is composed of ProjectMan, a primary component; MySQL and MyCRM, its composition constraints; and MyVCS a component that provides version control management functionalities and represents a possible composition of ProjectMan. The other two relations bound, respectively, MyCRM and MyVCS with their composition constraints. The functional part of the second composition plan is composed of one relation because none of the composition constrains and possibilities of MyProject has a composition constraint.

```
1  MPC: Matched Primary Components;
2  MPCC: Matched Primary Components Constraints;
3  MSC: Matched Secondary Components;
4  CP: List of composition plans;
5  cp: A composition plan (List of relations);
6  Relation, Rel: List of components involved in a
7   relation;
8
9  for (int i=0, i<MPC.size,i++)
10 {
11     Relation.add(MPC.get(i));
12     if(MPCC.get(i).size !=0)
13     {
14         for(int q=0, q<MPCC.get(i).size, q++)
15         {
16             Relation.add(MPCC.get(i).get(q));
17         }
18     }
19     CMB=all possible combinations meeting all user's
20     secondary functionalities from MSC.get(i)
21     if (CMB.isEmpty())
22     {
23         cp.add(Relation);
24         cp=verifConstraints(cp);
25         CP.add(cp);
26     }
27     else
28     {
29         Rel=Relation;
30         for each combination cmb from CMB
31         {
32             Relation.add(cmb);
33             cp.add(Relation);
34             cp=verifConstraints(cp);
35             CP.add(cp);
36             Relation=Rel;
37         }
38     }
39     Relation=new(list);
40 }
41
```

```
42 composition_plan verifConstraints(composition_plan cp)
43 {
44  int l=0; int nb_relation=1;
45  while(l<nb_relation)
46  {
47         for(int k=1, k<cp.get(l).size, k++)
48         {
49             if(getConstraints(cp.get(l).
50                 get(k)) not null)
51             {
52                 Relation=new(list);
53                 Relation.add(cp.get(l).get(k));
54                 for(int m=0; m<getConstraints(cp.get(l).
55                     get(k)).size, m++)
56                 {
57                     Relation.add(getConstraints(cp.get(l).
58                   get(k)).get(m));
59                 }
60                 nb_relation++;
61                 cp.add(Relation);
62             }
63         }
64         l++;
65  }
66  return(cp);
67 }
```

Listing 4 Composition plan generation algorithm

3.3.5 IaaS discovery phase

This phase selects the IaaS necessary to deploy the generated composition plans. IaaS available in the service repository are those upon which the orchestration tool can deploy components. A SPARQL query (Listing 5) is automatically generated to select IaaS (line 2) meeting user's preferences: preferred location (line 3) and preferred deployment provider (line 4).

```
1  SELECT  ?x WHERE {
2  ?x usdl:hasClassification "IaaS".
3  ?x gr:location  "Europe".
4  ?x gr:name  "Amazon".}
```

Listing 5 SPARQL query for IaaS selection

The discovered IaaS are ranked according to QoS requirements (service ranking is described in Sect. 3.3.6). The one with the highest rank is selected. The price of each previously generated composition plans using the selected IaaS is estimated. If the cost of a composition plan exceeds the maximum cost set by the user, the corresponding composition plan is excluded. If no composition plan remains, the IaaS having the next better rank is selected. Else, the functional part of the remaining composition plans is ranked. The configuration file of the orchestration tool is automatically updated to set the deployment environment upon the selected IaaS.

In the running scenario, the user prefers Amazon in Europe for hosting needs. The latter allows to deploy the two generated composition plans without exceeding the maximum cost set by the user. In Fig. 4, Amazon Europe has been added as a deployment part to the composition plans.

3.3.6 Composition plans ranking and selection phase

Composition plan's services are ranked according to user's QoS requirements and thanks to a "history of service invocation." The latter provides the QoS parameters describing the services according to their previous invocations. We consider

that the history of service invocation is provided by an independent third-party service as now we see the proliferation of cloud services comparison Web sites. Cloud armor [31] and Cloudorado [32] provide such third-party evaluation. The former provides a dataset of cloud services consumers' QoS ranking (availability, response time, ease of use, etc). The latter provides a comparison of cloud providers in terms of SLA level, price, and features.

Two scenarios are available for QoS parameters: (1) the highest the value of the QoS parameter is, the better the service is (e.g., availability), and (2) the lowest the value of the QoS parameter is, the better the service is (e.g., response time). R_{upper} and R_{lower} are, respectively, the ranks regarding these two kinds of QoS parameters. Let S_i be a service and Q_j be a QoS parameter.

$$R(S_i, Q_j) \begin{cases} R_{\text{upper}} = \frac{Val(S_i, Q_j)}{Max(Q_j)} * \text{Coefficient} \\ R_{\text{lower}} = \left(1 - \frac{Val(S_i, Q_j)}{Max(Q_j)}\right) * \text{Coefficient} \end{cases} \quad (1)$$

where:

- Val is the value of the QoS parameter for a given service.
- Max is the maximum value of the QoS parameter among all services supplying the same functionality.
- And, Coefficient is the weight previously assigned to the QoS parameter by the user.

Let $R(S_i)$ be the global ranking regarding the whole QoS parameters for S_i.

$$R(S_i) = \sum_{j=1}^{m} R(S_i, Q_j) \quad (2)$$

The rank associated with each possible composition plan is calculated as the average rank of the components involved in it (Eq 3). The plan that has the highest rank is selected.

$$R(\text{CompositionPlan}) = \frac{\sum_{i=1}^{NB} R(S_i)}{NB} \quad (3)$$

Where *NB* is the number of components involved in the composition plan.

3.3.7 Business application configuration phase

Some components can be configured according to user's preferences. For the selected composition plan, several Web interfaces are automatically displayed to the user according to the selected components and their configurable parameters so that she can personalize the generated application with specific details related to its business. For example, the user can personalize the component with the name and logo of her organization. She can also introduce username and password of the administrator and so on. From these information (introduced by the user via the Web form), the orchestration dedicated scripts are automatically created to configure the application.

3.3.8 Business application deployment phase

This phase consists of deploying the highly ranked composition plan that was generated in the composition plans generation phase. We generate a high-level script deploying the components and considering the relations between them. The script is dedicated to the cloud orchestration tool used ("Juju" in our implementation) in MADONA system for managing components deployment, configuration, and composition. It is generated as illustrated in Listing 6. For each relation from the selected composition plan, the first component is deployed. Then, each other component of the relation is deployed and related to the first component (lines 13–17). Finally, the first component of the first relation is exposed (line 20) to allow the user access the deployed application.

```
1  Input: Plan (represents the selected plan)
2  Output: script (deployment script)
3
4  for (int i=0, i<Plan.size, i++)
5  {
6     for (int j=0; j<Plan.get(i).size; j++)
7     {
8        if(Plan.get(i).get(j) has not been deployed yet)
9        {
10          script=script+"Juju deploy "
11          +Plan.get(i).get(j);
12       }
13       if (j>1)
14       {
15          script=script+"juju add-relation "
16          +Plan.get(i).get(j)+" "+Plan.get(i).get(0);
17       }
18    }
19 }
20 script=script +"juju expose "+Plan.get(0).get(0);
```

Listing 6 Deployment script generation algorithm

Listing 7 illustrates the automatically generated Juju dedicated command lines that deploy the selected composition plan. For each component deployment, we take into account its minimal resource requirements (line 1 of Listing 7).

```
1  juju deploy --constraints "cpu-cores=4 mem=1G" projectMan;
2  juju deploy mysql;
3  juju add-relation MySQL projectMan;
4  juju deploy MyCRM;
5  juju add-relation MyCRM projectMan;
6  juju deploy MyVCS;
7  juju add-relation MyVCS projectMan;
8  juju add-relation MySQL MyCRM;
9  juju deploy EmployMan;
10 juju add_relation EmployMan MyCRM,
11 juju add-relation MySQL MyVCS;
12 juju expose projectMan;
```

Listing 7 The generated deployment script for the "project management" scenario

Redeployment can occur if the user does not validate the resulted business application after the tests have occurred. In this case, the allocated resources for the previous deployment are released and another composition is deployed.

3.3.9 Tests and validation phase

The validation is done by the user after testing the deployed business application. Two types of tests are considered: performance and conformity. The former is done automatically using a testing tool such as Gatling [33]. The latter is done by the user who compares her requirements to the resulted business application. After tests, the user submits to the system her validation (satisfaction or dissatisfaction) regarding the deployed application. If the user is unsatisfied, another composition is deployed, other tests are performed, and the user has to notify her validation. This cycle is repeated until the user satisfaction is achieved or no other composition is possible.

4 Implementation

To validate and evaluate our approach, we implemented a system for MADONA as a Web application. A video of the system is available at liris.cnrs.fr/hind.benfenatki/demo.mp4. We chose Grails [34] as a framework that allows the development of applications following the Model, View, Controller (MVC) pattern. MADONA is based on an orchestration tool for managing the deployment of application components of a composition plan. We chose Juju as an orchestration tool; it allows component deployment, configuration, and composition using high-level scripts which is more suitable for automating the deployment phase. Furthermore, Juju store offers components that have the necessary support for our implementation. We use, respectively, Jena API and Jena-arq API to model and query .rival and .usdl files.

The system is deployed on top of a Dell machine, 1.80 GHz with 16 GB memory, running Windows 8.1. We used VirtualBox to install two guest Ubuntu VMs which get 4 GB of memory and 80 GB of disk in dynamic allocation. We installed MADONA system on one VM and Juju environment on the second. Juju VM hosts the Web application generated by MADONA. In fact, Juju environment simulates a cloud environment upon which several Linux containers are instantiated. Necessary components are deployed on these Linux containers. The two VMs communicate using OpenSSH.

Figure 5 illustrates the architecture of MADONA system. It is composed of three levels. *The interface level* is responsible for communicating with the user. A controller routes data inputs (introduced by the user via Web interfaces) between various Java classes. Five views have been created. "NewProject" view to introduce requirements for a new project. "Config" view to introduce configuration parameters. "AddCharm" view to upload new charms. "AddDescription" view to describe new added components. And "Home" view to display the status of deployed components.

The application level allows to generate and deploy composition plans. First, user's requirements are translated into .rival file ("RivalGen" class). A SPARQL query is generated and launched on the .usdl descriptions of services, and composition plans are generated ("DiscoCompo" class). The latter are ranked and ordered using a bubble sort and their identification is stored in a text file ("RankingCalcul" class). The user introduces her configuration parameters ("Config" view) for the composition plan with the highest rank. Configuration and deployment scripts are generated ("DeployConfigScriptGen" class) and sent to Juju server using SSH for execution (respectively, "AutoConfig" and "AutoDeploy" classes). The user is then sent back to the "Home" page where the matched application is made available with the status "is being deployed." An auto refresh of the Web page insures that a script is sent and executed every thirty seconds to obtain the status of Juju ("IPFind" class). The latter is analyzed using java String tools and when the IP address appears a link is provided to that application. To allow integrating new components to Juju store, three other classes have been implemented. "UploadCharm" which allows to transfer charms archive of a new component to Juju VM. "USDLGen" class generates .usdl descriptions in turtle format, from a given description introduced via the "AddDescription" view. For each new component, "QoSToXML" class generates randomly QoS values within a predefined interval for each QoS parameter and stores them in an XML file. The QoS XML files are used in the composition plans ranking phase.

The service level consists of USDL descriptions, QoS XML files, and distributed components packages.

5 Comparing our system to Bitnami and Juju

We evaluate MADONA system by comparing the provisioning of MediaWiki, WordPress, and the running scenario using: (1) Bitnami IaaS [35], (2) orchestration tool Juju, (3) MADONA system, and (4) local deployment in Ubuntu machine.

Bitnami allows to deploy ready and static cloud applications in a simple and automated manner. The user has to select the appropriate application, deployment provider and location, operating system, server type, disk size, application options such as login and password of the application, development options to include the installation of Web servers, and application properties such as language and nickname. These inputs have default values to allow the user to deploy her application easily.

Figure 6 shows the setup time of the Juju environment according to the different phases: installation, configuration,

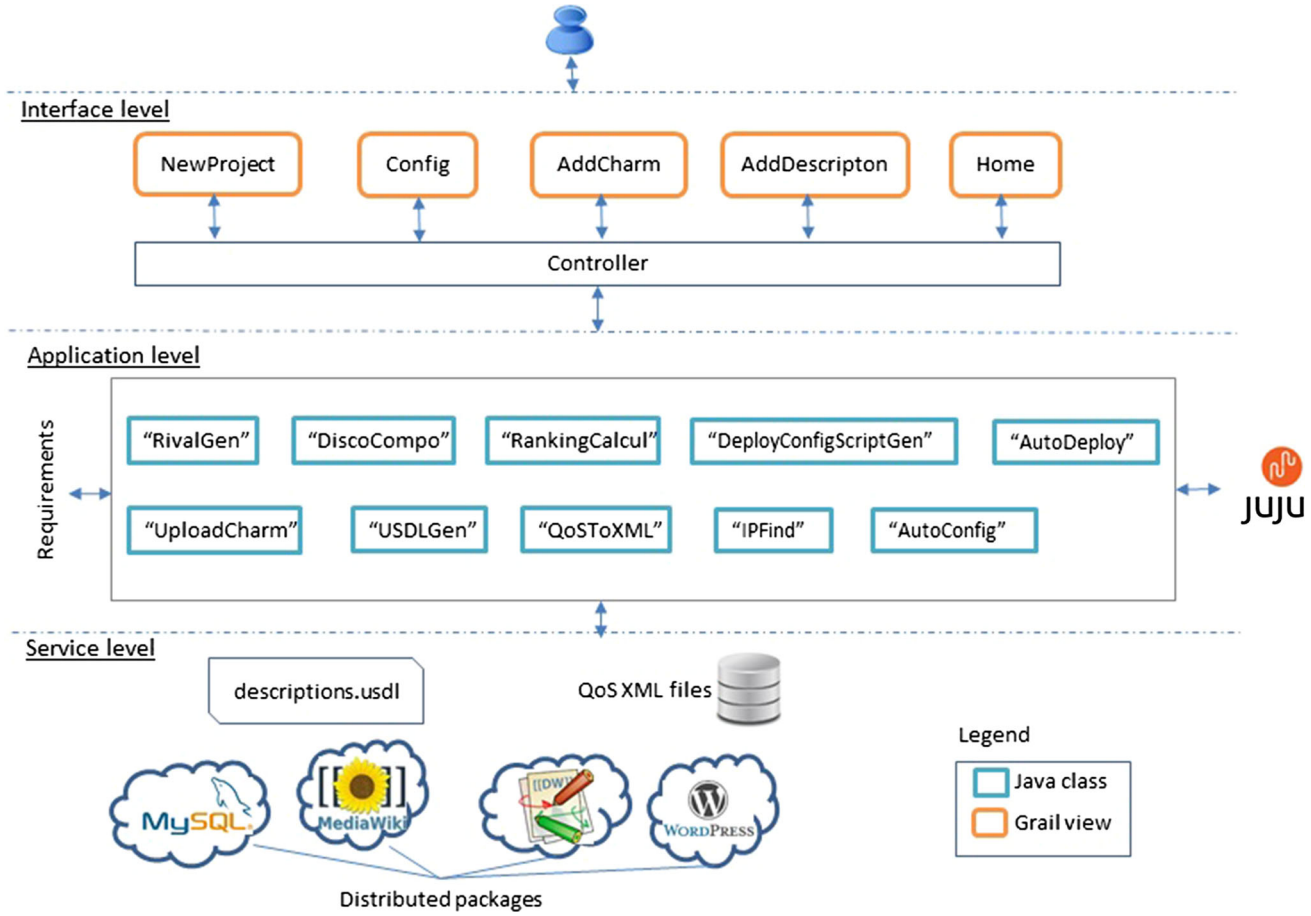

Fig. 5 Architecture of MADONA system

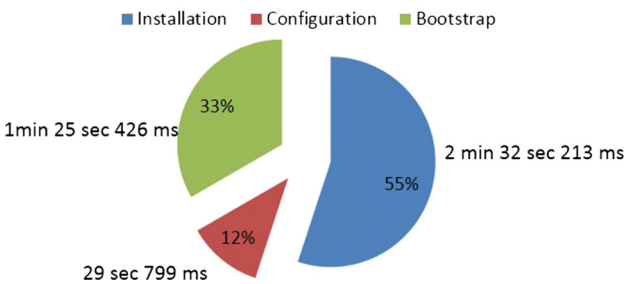

Fig. 6 Juju environment setup time

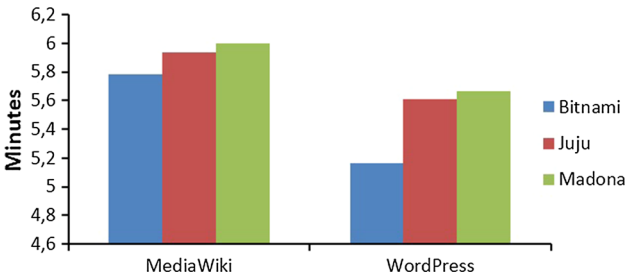

Fig. 7 Provisioning time of MediaWiki and WordPress

and bootstrap. Bitnami, MADONA, and local deployment do not require any environment installation.

Figure 7 shows the provisioning time of MediaWiki and WordPress scenarios. Each scenario is composed of three components. The choice to evaluate provisioning time using MediaWiki and WordPress scenarios is guided by the fact that the components involved are available in Bitnami and Juju. The discovery and deployment times are hard to evaluate for Juju and local deployment due to the limited control over any manual work. In fact, the discovery process varies depending on how it is done, what selection criteria are, and so on. And the deployment time varies depending on how familiar the user is with this kind of installation. The provisioning of MediaWiki and WordPress using MADONA consumes more time than Bitnami. In fact, an increase of 13 s (+3 %) is observed while provisioning MediaWiki. The purpose of evaluating provisioning time with Bitnami and MADONA is to show that MADONA provisions applications in a satisfactory time in comparison with an industrial solution. The provisioning of MediaWiki and WordPress using MADONA consumes more time than Juju (an increase of 3 s (+1%) is observed while provisioning MediaWiki). How-

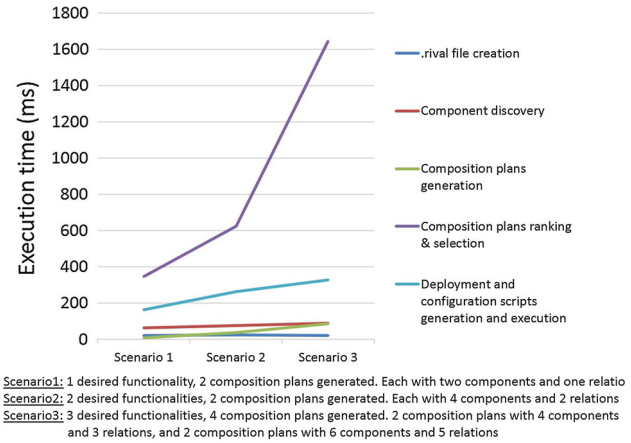

Fig. 8 Execution time of MADONA's phases using three scenarios

ever, MADONA automates the phases before the deployment and is based on Juju. We choose to evaluate the provisioning time instead of resource consumption to analyze efficiency through computational cost for the following reasons:

- Bitnami is an industrial platform, so it is hard to compare MADONA with Bitnami because we do not have any information related to resource allocation and consumption for the provisioning using Bitnami.
- Optimizing the computational cost through provisioning time and resource consumption is out of the scope of this work.

Despite a larger provisioning time using MADONA, it allows to compose components on the fly and automatically meeting user's requirements. MADONA reduces the technical knowledge needed to provision any cloud-based component-oriented business applications (Figs. 9, 10). In fact, each phase of MADONA is fully automated, and the user's requirements are expressed in a high-level regarding the technical details (in terms of functionalities, QoS requirements, and cost and deployment preferences). Conversely, Bitnami deploys only ready applications, and Juju requires user's intervention in the selection of the required components and in the deployment script generation.

Figure 8 illustrates the execution time of MADONA's phases following three scenarios varying the number of (1) desired functionalities, (2) generated composition plans, and (3) components and relations in each composition plan. The phases consuming more time are those manipulating files. We observe a remarkable increase in composition plans ranking time as the size of the obtained results increases (in terms of number of generated composition plans, relations, and components involved). This is due to the fact that the ranking phase queries one QoS XML file per component involved in the generated composition plans.

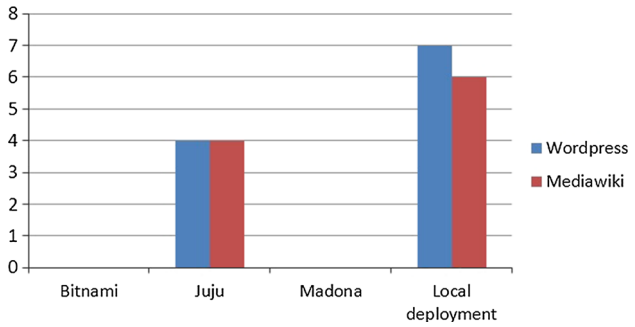

Fig. 9 Quantity of script lines needed to deploy MediaWiki and Wordpress

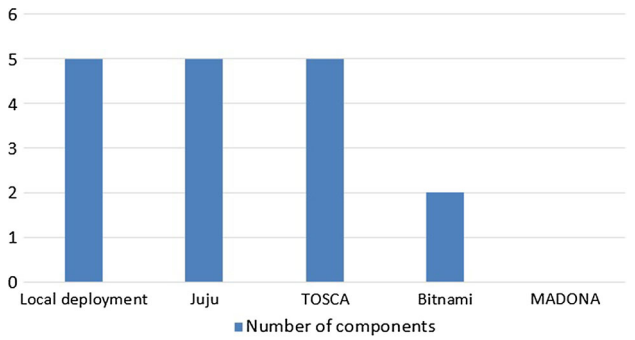

Fig. 10 Components the user has to know to provision our running scenario following different approaches

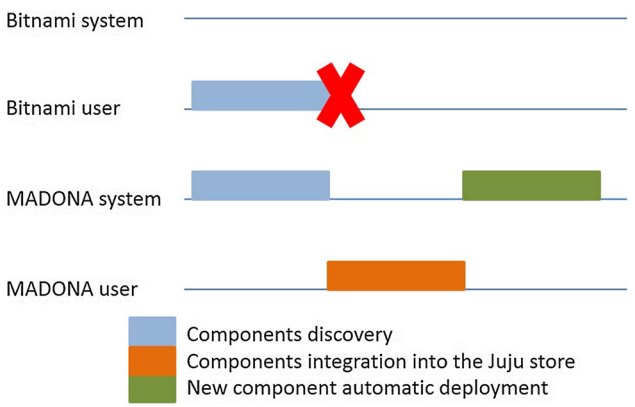

Fig. 11 MADONA Versus Bitnami when existing components do not meet user's requirements

Out of Fig. 9, we observe that the user has to deploy the application components using scripts for orchestration tool Juju and local deployment in Ubuntu machine, while in Bitnami IaaS and MADONA system, the scripts are generated automatically. The deployment on an IaaS (like Bitnami) is easy to complete, but this is done in a preconfigured VM upon which the primary component and its composition constraints are deployed on a same machine. The composition

Table 1 Bitnami versus Juju versus MADONA

Phase	Bitnami	Juju	MADONA
Requirement elicitation	Considers only primary functionalities. The user chooses explicitly the application to deploy. The user has to choose from a set of existing applications	Considers only primary functionalities. The user chooses explicitly components to deploy and compose. The user has to choose from a set of existing components	Considers primary and secondary functionalities. Needed components are described in terms of functionalities or by their name
Application components discovery	No discovery since the user chooses the needed application	No discovery since the user chooses the needed components	Components are chosen based on required functionalities or required components. Discovery concerns needed components and their composition constraints and possibilities
Integration of new components	Is possible only by the provider. The user has, however, the possibility to import her VMs from Amazon. Without the description of the newly integrated components, the latter cannot be automatically connected to other components in future compositions	The user has the ability to integrate new components to the Juju charms store by integrating scripts for deployment, configuration and composition with other components	The user has the ability to integrate new components to the service repository through the integration of components descriptions. She also integrate to the Juju charms store scripts for deployment, configuration and composition with other components. The newly integrated components are automatically taken into account in future compositions thanks to their description
Composition plans generation	Does not exist. Composition possibilities are not taken into account. The user selects the application to deploy	Does not exist. Composition constraints are known but not taken into account automatically. The user selects the components to deploy and to compose	Composition constraints and possibilities are known (from component's description) and taken into account automatically to generate composition plans. The user does not have to check whether components work well together
IaaS discovery	Does not exist. The user must choose an IaaS for the deployment. The user has to know (or inquire about) prices of a given VM in a given IaaS	Does not exist. The user must choose an IaaS for the deployment. The user has to know (or inquire about) prices of a given VM in a given IaaS	An IaaS is selected automatically according to user preferences if she provides them (preferred location and provider), QoS requirements, and deployment cost to not exceed
Composition plans ranking and selection	Not taken into account	Not taken into account	Composition plans are ranked to select the best one (in terms of QoS)
Business application configuration	Through a Web interface and specific to each application	Through configuration scripts and specific to each application component	Through a Web interface and specific to each application component
Business application deployment	A type of VM should be selected for the deployment of the application. A default value exists to reduce the required knowledge ⇒ The user has to know the minimal configuration allowing the good functioning of the application	A type of VM should be selected for the deployment of components. It has to be mentioned in Juju configuration file. A default value exists to reduce the required knowledge ⇒ The user has to know the minimal configuration allowing the good functioning of the component. The user creates the deployment script	Minimal required resources (CPU and memory) are known (from component description as illustrated in Listing 1) allowing to deploy the component in a VM having sufficient resources. This is taken into account automatically. Deployment script is automatically generated and executed

Table 2 Local deployment versus using Juju versus using Bitnami versus using MADONA

System	Local deployment	Using Juju	Using Bitnami	Using MADONA
Prerequisite	Web server Data base server Components' packages	Juju server	Web browser	Web browser

Table 3 MADONA versus the related work

Work	Approach	Cloud level	Comments
SOSDC [13]	Service-Oriented	SaaS, PaaS, IaaS	Platform specific
TOSCA [4]	Package oriented	SaaS, IaaS	Topology for the deployment of a cloud application describing the structural description of the application
Juju [5]	Orchestration tool	IaaS	Allows to deploy components using charms
Bitnami [35]	Application hosting platform	SaaS	Allows to deploy supplied applications
MADONA	Component-Oriented	SaaS, IaaS	Complete method for automatic and dynamic cloud application provisioning

is not done dynamically, but rather the several compositions have to be known and scripted in a static way.

Figure 10 illustrates the components that a user has to be aware of to provision an application with two desired functionalities and three composition constraints (such as the first composition plan generated for the running scenario). For both local deployment, Juju, and TOSCA, the user has to know all the components involved in the desired application. Using Bitnami, the user has not to worry about the composition constraints. However, Bitnami does not take into account composition possibilities when deploying applications. So, the two components meeting the two functionalities are not composed. Using MADONA, the user does not have to know any component of the desired application.

Figure 11 compares Bitnami and MADONA when available components do not meet user's functional and nonfunctional requirements. While MADONA allows the user to enrich the repository of services by adding external components, Bitnami just allows users to import their EC2 instances. Using MADONA, new components are automatically integrated into the service repository and used (discovered, composed and deployed) in future provisionings.

As a first conclusion, MADONA system "is close" to Bitnami in the fact that the user does not need to write any script to deploy the needed application and does not require any preinstallation. However, these two systems differ essentially in the necessary technical knowledge when provisioning of the desired application (Fig. 10; Table 1), and in the provisioning lifecycle as illustrated in Table 1. Differences between MADONA and Bitnami are explained in the next Section.

6 Discussion

We have cited various cloud application development and deployment approaches. Each covers SaaS, PaaS and/or IaaS levels.

Tables 1 and 2 highlight the common and varied properties between Bitnami, Juju, local deployment, and MADONA. Even if the deployment of the desired application is automated for Bitnami and MADONA systems, the application construction is different. In fact, with Bitnami, the user chooses the application to deploy, the IaaS upon which it will be deployed, and the necessary VM type. The composition constraints are automatically taken into account in a static way, i.e., Bitnami does not compose application components on the fly following users' requirements. Furthermore, Bitnami does not take into account composition possibilities in application deployment. Using MADONA, composition constraints and possibilities management are done automatically and dynamically making the process generic and enrichable. Furthermore, the discovery process using MADONA reduces technical knowledge because users are asked to supply the information about the needed functionality instead of the application name, and they do not have to select IaaS provider and VM types.

Table 3 illustrates the comparison between the related work and our approach according to the following criteria: cloud level covered and used approaches. It appears that the related work is focusing on a special issue such as deployment of cloud applications [4,5], or development environment [13]. MADONA provides a requirement vocabulary for cloud applications; extends Linked USDL to describe the composition constraints and composition possibilities in order to make the composition plan generation automatic and

dynamic; automates the deployment process; uses orchestration tools to deploy and manage constraints and possibilities between components in a dynamic way.

7 Conclusion

In this paper, we presented MADONA—a Method for AutomateD prOvisioning of cloud-based component-oriented busiNess Applications—that reduces the technical burden on users of knowing cloud application provisioning. MADONA covers application provisioning lifecycle from requirement elicitation to validation phases. It is iterative and adaptive to user needs allowing to deploy several applications until user's requirements are met. We also defined RIVAL—a RequIrements VocAbuLary—for describing users requirements in order to provision a cloud business application. To automate the discovery of components, we extended Linked USDL to track the relations that a component *can* and *must* have with peers such as composition constraints and possibilities. We use "Juju," a cloud orchestration tool, which facilitates the deployment and management of dependencies of components. Component dependency management is done dynamically making the process generic and the repository of services enrichable. MADONA system has been developed and tested following a running scenario.

As part of our ongoing work, we intend to integrate a discovery approach allowing to query distributed repositories of services. We also plan to integrate semantic matching in the components discovery phase. Also, a negotiator module will be added to allow the system to negotiate user preferences, while composition plans are generated.

Acknowledgements The authors would like to thank Professor Jorge Cardoso for commenting earlier versions of the manuscript. They would also like to thank the anonymous reviewers for their constructive feedback.

References

1. Tsai WT, Sun X, Balasooriya J (2010) Service-oriented cloud computing architecture. In: Seventh international conference on information technology: new generations (ITNG). IEEE, pp 684–689
2. Amazon CloudFormation (2016) https://aws.amazon.com/fr/cloudformation/
3. Heat (2016) https://Wiki.openstack.org/Wiki/Heat
4. Binz T, Breitenbücher U, Kopp O, Leymann F (2014) TOSCA: portable automated deployment and management of cloud applications. In: Bouguettaya A, Sheng QZ, Daniel F (eds) Advanced web services. Springer, New York, pp 527–549
5. Juju (2016) https://juju.ubuntu.com/
6. Linked USDL (2013) http://www.linked-usdl.org/
7. Cardoso J (2013) A unified language for service description: a brief overview. http://www.issip.org/2013/04/26/a-unified-language-for-service-description-a-brief-overview/
8. Pedrinaci C, Cardoso J, Leidig T (2014) Linked USDL: a vocabulary for web-scale service trading. The semantic web: trends and challenges. Springer, Berlin, pp 68–82
9. Thoma M, Antonescu AF, Mintsi T, Braun T (2013) Linked services for enabling interoperability in the sensing enterprise. Enterprise interoperability. Springer, Berlin, pp 131–144
10. Barros A, Oberle D (2012) Handbook of service description: USDL and its methods. Springer, Berlin
11. Gu Q, Lago P (2009) Exploring service-oriented system engineering challenges: a systematic literature review. Service oriented computing and applications. Springer, Berlin
12. Ardagna D, Di Nitto E, Casale G, Petcu D, Mohagheghi P, Mosser S, Matthews P, Gericke A, Ballagny C, D'Andria F, Nechifor CS, Sheridan C (2012) Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In: 4th international workshop on modeling in software engineering. IEEE, pp 50–56
13. Sun H, Wang X, Zhou C, Huang Z, Liu X (2010) Early experience of building a cloud platform for service oriented software development. In: IEEE international conference on cluster computing workshops and posters (CLUSTER WORKSHOPS). IEEE, pp 1–4
14. Zhou J, Athukorala K, Gilman E, Riekki J, Ylianttila M (2012) Cloud architecture for dynamic service composition. Int J Grid High Perfor Comp (IJGHPC) 4:17–31
15. OpenStack open source cloud computing software (2014) https://www.openstack.org/
16. OASIS-Advanced open standards for the information society (2014) https://www.oasis-open.org/
17. TOSCA Language (2014) http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html#_Toc356403635
18. Amazon elastic compute cloud (2014) http://aws.amazon.com/fr/ec2/
19. HP Cloud (2014) http://www.hpcloud.com/
20. Windows Azure (2014) http://azure.microsoft.com/fr-fr/
21. Linked data - connect distributed data across the web (2013) http://www.linkeddata.org/
22. Cardoso J, Binz T, Breitenbücher U, Kopp O, Leymann F (2013) Cloud computing automation: integrating USDL and TOSCA. In: Conference on advanced information systems engineering. Springer, Berlin
23. Nguyen DK, Lelli F, Papazoglou MP, Van den Heuvel WJ (2012) Issue in automatic combination of cloud services. In: IEEE 10th international symposium on parallel and distributed processing with applications (ISPA). IEEE, pp 487–493
24. Juju charms store (2016) https://jujucharms.com/store
25. Linked USDL modules (2013) https://github.com/linked-usdl
26. Egyed A, Grunbacher P (2004) Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help. IEEE Softw 21:50–58
27. Hausmann JH, Heckel R, Taentzer G (2002) Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation.In: 24th international conference on software engineering. ACM, pp 105–115
28. Easterbrook S (1994) Resolving requirements conflicts with computer-supported negotiation. Requirements engineering: social and technical issues. ACM pp 41–65
29. Vu LH, Hauswirth M, Aberer K (2005) Towards P2P-based semantic web service discovery with QoS support. In: International conference on business process management. Springer, pp 18–31
30. Nayak R, Lee B (2007) Web service discovery with additional semantics and clustering. In: International conference on web intelligence. IEEE, pp 555–558
31. Cloud Armor (2016) http://cs.adelaide.edu.au/~cloudarmor/ds.html
32. Cloudorado (2016) https://www.cloudorado.com/
33. Gatling: Load testing tool (2014) http://gatling-tool.org/
34. Grails framework (2015) https://grails.org/
35. Bitnami: Cloud hosting (2014) https://bitnami.com/