International Journal of Cooperative Information Systems
Vol. 20, No. 3 (2011) 307–356
© World Scientific Publishing Company
DOI: 10.1142/S0218843011002250

MOVING APPLICATIONS TO THE CLOUD: AN APPROACH BASED ON APPLICATION MODEL ENRICHMENT

FRANK LEYMANN*, CHRISTOPG FEHLING,
RALPH MIETZNER and ALEXANDER NOWAK

Institute of Architecture of Application Systems
University of Stuttgart, Stuttgart 70569, Germany
**frank.leymann@iaas.uni-stuttgart.de*

SCHAHRAM DUSTDAR

Distributed Systems Group
Vienna University of Technology
Wien 1040, Austria
dustdar@infosys.tuwien.ac.at

In this paper we describe a method and corresponding tool chain that allows moving an application to the cloud. In particular, we support to split an application such that various parts of it are moved to different clouds. This split can be done manually or by support of optimization algorithms. The split application is then automatically provisioned in the different target clouds. A metamodel for such applications supporting the proposed method is introduced. The architecture of a supporting tool is described. Experiences from the usage of the proposed method are reported.

Keywords: Application modeling; metamodels; cloud computing.

1. Introduction

Today, many companies consider moving entire applications or parts of them to the cloud.[1–5] Applications today are often composite, multi-tier applications, consisting of application components such as UIs, services, workflows and databases as well as middleware components such as application servers, workflow engines and database management systems. When moving such a composite application into the cloud, decisions must be made about putting which tier and even which component of such an application to which cloud.[6] Drivers for these decisions include functional properties of a cloud such as the possibility to run a specific required middleware and non-functional properties such as data privacy, cost and offered quality of service by a specific cloud provider. European enterprises, for example, face difficulties in putting customer-relevant data into a cloud that has resources that are physically outside the European Union. They may, however, opt to put other parts of an application that are not dealing with customer-relevant data into an overseas cloud that might be cheaper or offers superior quality of service.

Effectively, moving an application to the cloud is a rearrangement of the application's deployment topology in which component dependencies are captured. Such a rearrangement of an application is often not only based on criteria like latency and data transfer, as investigated in distributed systems research in the past, but also on criteria such as data privacy, legislative compliance or trust, for example. Thus, an approach is needed to support splitting and scattering (i.e. rearranging) applications in a generic way to support a variety of reasons for splitting.

The *general problem to be solved* is then (i) how to rearrange the components of a multi-tier, multi-component application into disjoint groups of components, such that (ii) each such group can be provisioned separately to different clouds while preserving the desired properties of the whole application — we refer to this problem as the *Move-to-Cloud* problem.

In this paper, we formally transform the Move-to-Cloud problem into a graph partitioning problem and use existing optimization algorithms such as simulated annealing to optimize the distribution of components between different clouds. The main contribution of this paper is thus not a novel optimization algorithm, but the *methodology* and a corresponding *tool chain* that allows application developers and architects to (i) model their application components and properties and (ii) define relevant criteria for the splitting. This is done using a variety of diagrams and models that capture the information relevant for the splitting. The annotated application models then serve as an input for the optimization algorithms which produce sets of component groups that can be moved into the same cloud. The presented tools are integrated with existing provisioning tools to automatically setup the components in the correct cloud.

One essential property of the presented approach is its *general applicability*, i.e. the approach does not depend on one cloud framework, virtualization technology or programming language, but gives general guidance on how to solve the Move-to-Cloud problem for a large variety of programming languages, virtualization technologies and clouds. Therefore, the presented approach is not limited to public clouds but is also suitable for private and hybrid clouds and can even be exploited (with limitations regarding elasticity) for the splitting of applications that are (partially) run in traditional datacenters.

The presented approach is based on requirements from two projects in concrete companies the authors have been involved in. These projects dealt with existing JEE applications as well as process-based service applications on the Web. The software stacks used in the projects have been corresponding JEE and SOA stacks including relational database systems, both, from commercial vendors as well as from open source vendors. In one project, the rearrangement of the application was based on trust criteria, the other project was focused on costs. The corresponding modeling of the applications as well as the provisioning in the target clouds have been prototypically realized based on the tools presented in the paper. Both projects split their applications across a public cloud and a private cloud, but different clouds have been used in the different projects.

The paper is structured as follows: Section 2 discusses the conceptual approach to move applications to the cloud and a running example is given; especially, a corresponding method and a supporting metamodel are presented. Core concepts underlying the presented method are formally defined in Sec. 3 and the problem of automatically deriving cloud distributions is presented as an optimization problem. The architecture of a prototypical tool suite supporting the proposed method is described in Sec. 4. Experiences in using the proposed method in a concrete use case are reported in Sec. 5. The presented approach is compared to related work in Sec. 6. Finally, Sec. 7 concludes the paper.

2. Conceptual Approach

In this section, we discuss the details of the proposed method called MOCCA (MOve to Clouds for Composite Applications), its metamodel and its underlying concepts. A running example is used to demonstrate the major steps of the method.

2.1. *First overview of the MOCCA method*

The proposed method assumes that for the application to be moved to the cloud three main artifacts will be provided: (i) an architecture model of the application, (ii) a deployment model of the application, and (iii) implementation artifacts such as virtual images of (parts of) the application. As an example for the application to be moved to the cloud, we exemplarily use a simple order system that is able to receive and evaluate a user's order request, process the order and finally make the results persistent (see Fig. 3). This sample application abstracts the kind of applications we dealt with in practice: it has a Web frontend, makes use of servlets and enterprise Java beans, and depends on a Web server, an application server, and a database system.

Covering the three main artifacts, the *architecture model* first describes the architectural components of the application (i.e. the "boxes" of the diagram) and their relations (i.e. the "arrows" of the diagram). Note that the granularity of the specified components has an impact on the flexibility and quality of the split of the application into groups that are provisioned in different clouds (see Sec. 2.7). The *deployment model* specifies the runtime containers required by the application and which component of the architecture is hosted by which of the containers. Furthermore, deployment relevant parameters must be indicated that will be needed at provisioning time at the latest. The *implementation artifacts* of the application encompass installable units of the application, like executable or virtual images of (parts of) the application. But it may contain more than that, and the content of the virtual image has impact on quality of the resulting installation in the cloud (see Sec. 2.8).

Based on the first two artifacts a fourth artifact is derived called a *cloud distribution* (see Sec. 3.1). A cloud distribution is a set of architectural components of the application that are to be moved to the same cloud. As shown later, a cloud distribution can be specified manually or it can be derived automatically. An automatic derivation of a cloud distribution requires specifying additional information (so-called "labels") with the architecture diagram (see Sec. 3.2). Finally, the actual provisioning of the cloud distribution in the target clouds is performed based on the automatic creation of a fifth artifact called a *provision cluster* (see Sec. 3.1). During provisioning, actual values for the relevant deployment parameters indicated with the deployment model will be derived or enquired (see Sec. 4.3).

Before describing the MOCCA method in detail (see Sec. 2.6), we discuss the metamodel underlying the MOCCA method in the following Sec. 2.2. Next, the sample application of the simple order system is given in detail and modeled using the proposed metamodel at its architectural level (Sec. 2.3), at its deployment level (Sec. 2.4) as well as its provisioning and virtual image level (Sec. 2.5).

2.2. *The MOCCA metamodel and diagram types*

In Ref. 7, we propose a framework for provisioning customizable composite applications in the cloud. This metamodel has been adapted for the purpose of supporting MOCCA and is shown in Fig. 1. Note that only those attributes are shown and discussed here which are relevant in our context.

A customizable application is represented by an instance of the entity type `Application Template`. Such a template `consists of` one or more instances of the `Component` entity type. A component may `contain` other components. Amongst other attributes a component has a `Name` and a `Type`. The latter attribute has no fixed set of predefined values; for example, a component may be of type *Application Server*. A `Component` is `source of` as well as `target of` zero or more `Component Relation` entities. The relevant attribute of a `Component Relation` is its `Type` attribute indicating the semantics of the relation between the two associated components. Each component relation and each component `has` zero or more `Labels` which are specified as pairs of a `Name` and a `Value` attribute of the `Label` entity (the role of labels is described in Sec. 3.2).

Each component is `realized by` exactly one `Implementation`. The most important attribute of the implementation is the `Type` attribute. This attribute indicates the main manner or technological basis used to realize the implementation (e.g. whether it has been realized as a *BPEL* orchestration, or an *OVF* image etc.); for example, a `Component` of `Type` *Application Server* may be realized by an `Implementation` of `Type` *OVF*. If the implementation is of `Type` *External*, it points to its realization via an Endpoint Reference (EPR)[8]; if it is of `Type` *Provider Supplied*, the actual realization of the component will be provided at a later point in time by a particular provider (e.g. the provider has it already installed and as basis for the proper installation and deployment of new components). The middleware components in the practical exploitations of MOCCA had been of *Provider Supplied* implementation type to get experiences with middleware offered in the cloud; the

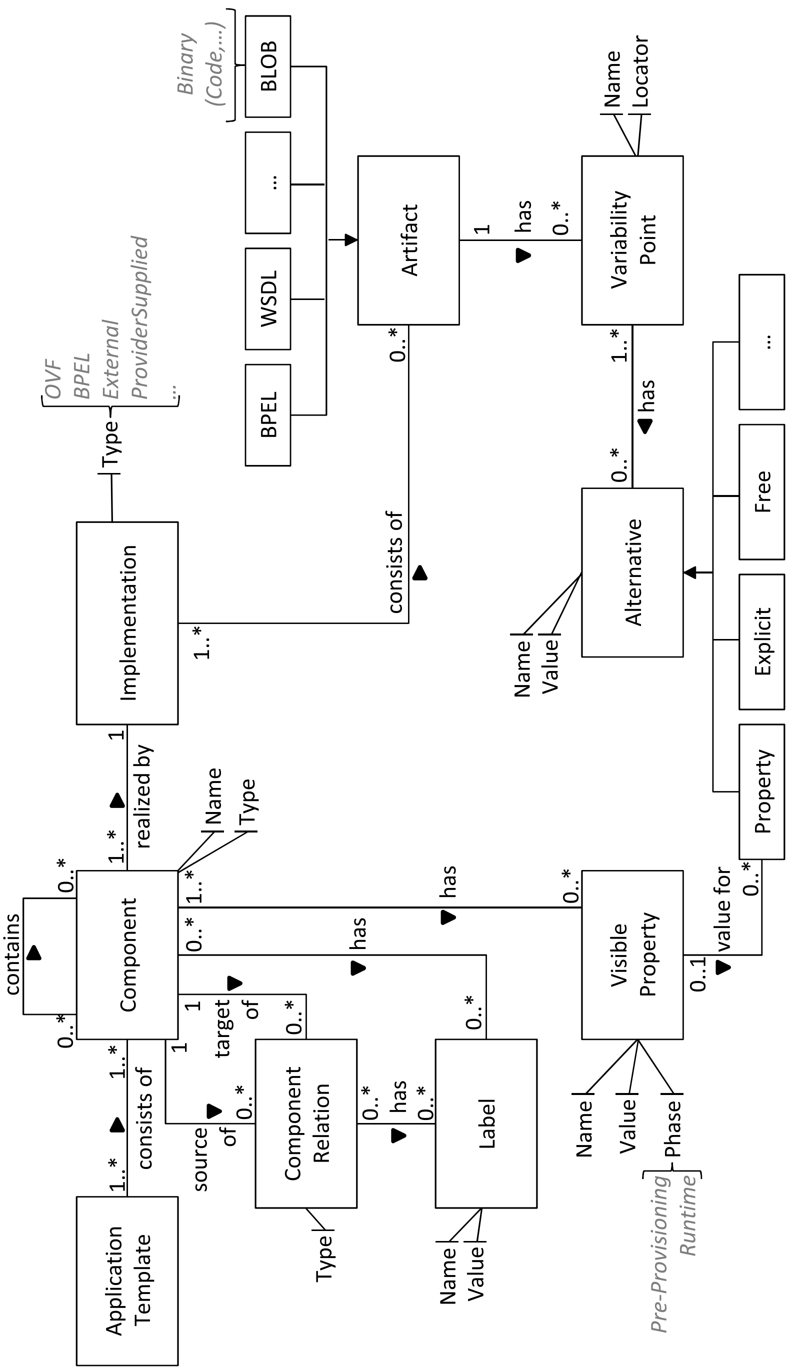


Fig. 1. Metamodel for composite applications.

implementations of application specific components had been of type *BPEL*, *WSDL* etc. An implementation consists of zero or more `Artifacts`. An artifact is the generalization of different kinds of artifacts like `BPEL` files, `WSDL` files, and so on up to `BLOBs` that contain binaries of actual code. For example, an `Implementation` of type *BPEL* consists of BPEL files (i.e. instances of the `BPEL` artifact), WSDL files (i.e. instances of the `WDSL` artifact) and other corresponding artifacts (e.g., deployment descriptors, . . .).

An artifact has zero or more `Variability Points`. A variability point has a `Name` and a `Locator` attribute. The latter attribute is used to point directly into the artifact to distinguish the piece within the artifact that may be overwritten; for example, a locator may be an XPath expression pointing to an operation name of a port type of a WSDL file. A variability point is associated with zero or more `Alternatives`. An alternative has a `Name` and `Value` attribute. When binding a variability point it is assigned a value of exactly one of the alternatives associated with the variability point. Thus, the set of alternatives associated with a variability point support users in customizing an application template by providing a list of potential values to choose from a variability point. There are multiple types of `Alternatives`. In our context `Explicit` alternatives, `Free` alternatives and `Property` alternatives are relevant. An `Explicit` alternative provides a pre-defined value that a user can select when binding a variability point. A `Free` alternative allows the input of an arbitrary value by a user to bind a variability point. `Property` alternatives point to a `Visible Property` of a Component.

A `Visible Property` is a property of a component that is made visible to the outside for the purpose of overwriting. A visible property has a `Name` and a `Value` attribute; for example, its `Value` can be an EPR under which its associated component can be reached. The `Phase` attribute of a visible property defines the point in time when it becomes available for overwriting. The two `Phases` relevant for this context are *Pre-Provisioning* (i.e. the component is not yet provisioned) and *Runtime* (i.e. the component is already running). In case a `Property` alternative points to a visible property the `Value` of this visible property serves as the `Value` of the `Property` alternative and is thus used to bind the associated variability point.

Figure 2 summarizes how the metamodel represents the various artifacts assumed by the MOCCA method are represented by the proposed metamodel. The corresponding metamodel elements are grouped by dashed lines, and the names of the corresponding artifacts are given in rectangles with rounded edges. `Components` and `Component Relations` of the metamodel are used to describe the "boxes" and "arrows" of the architecture diagram of an application (see Sec. 2.3 for an example). The metrical annotations of a "box" or an "arrow" of an architecture diagram used to automatically propose a cloud distribution of an application (see Sec. 3.2) are represented by the `Labels` associated with the `Component` representing the "box" or with the `Component Relation` representing the "arrow". At the topological level deployment models are represented by means of `Components` and the `contains` relationship between components: a container at the middleware level is represented

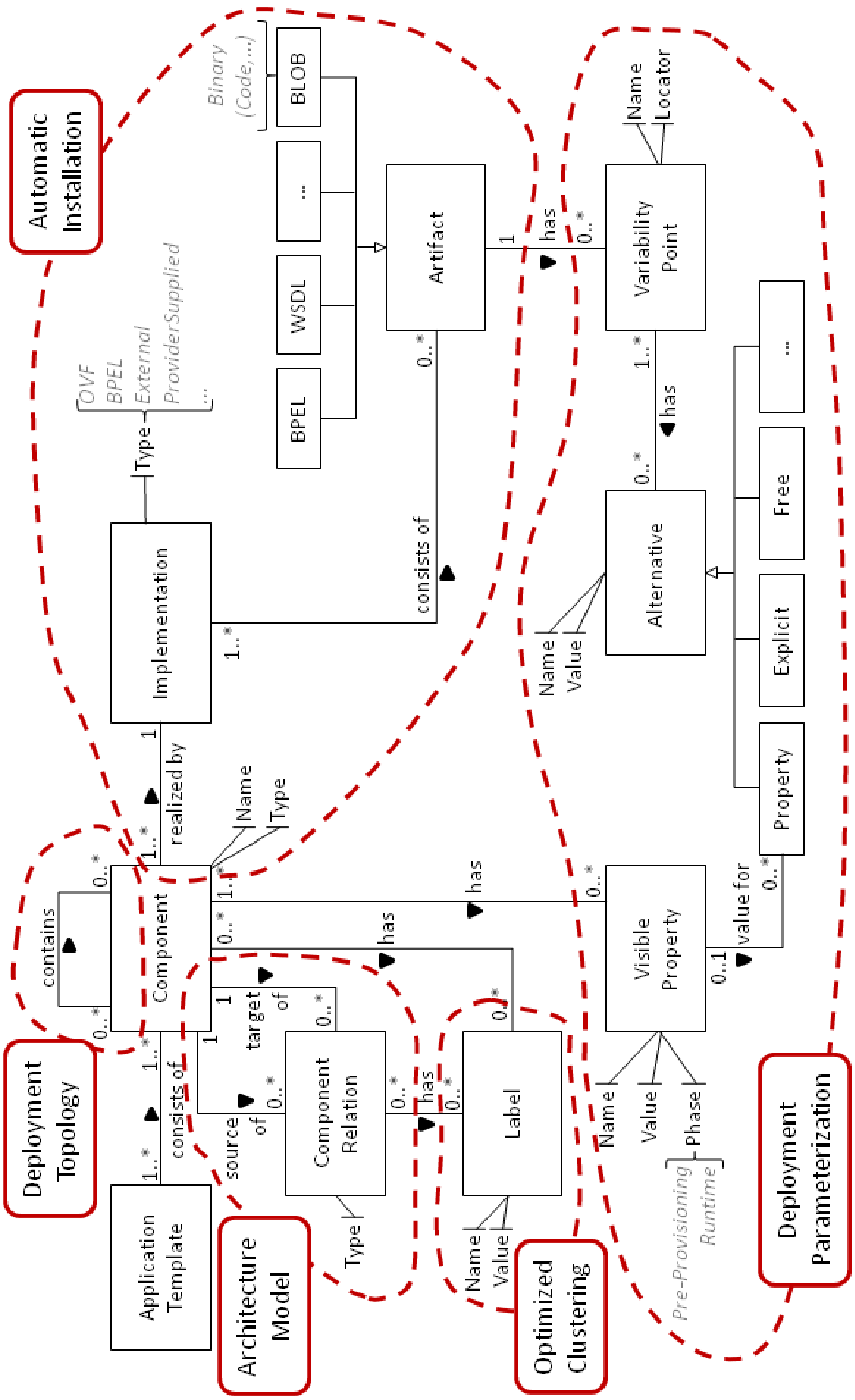


Fig. 2. Model types and metamodel.

as an instance of `Component` and `contains` all components it hosts (see Sec. 2.4 for an example). Beyond the topology of a deployment `Visible Properties` and `Variability Points` can be defined for the components of an application to support the specification of the parameterization aspects of a deployment (see Sec. 2.5 for an example) which will support an automatic provisioning of applications. To support an automatic installation of an application the `Implementation` and `Artifacts` of a component have to be defined. We employ a very generic metamodel for various reasons. First of all, this generic approach does not restrict the approach to a particular platform or programming language. By using a generic orthogonal variability model we allow all variability of an application to be expressed in one model. This variability can range from SLAs to functional variability. Having an orthogonal variability model is necessary as variability in one component (for example, the required availability of an application server) might depend on the binding of other variability points of other components (for example, the required availability of the whole application).

Our model allows importing the visible properties of other components in the model of an application template. This enables providers or middleware vendors to advertise the visible properties for a component (for example, an application server), that can then be imported into the model of an application that makes use of that application server thus allowing to reuse already modeled artifacts.

2.3. *Example — architecture level*

The application to be moved into the cloud is a simple order system; note again that the sample application is an abstraction of the applications we dealt with in practice, but it shows all the major aspects relevant to see how our method can be used in practice. Its architecture diagram is sketched in Fig. 3; as usual,

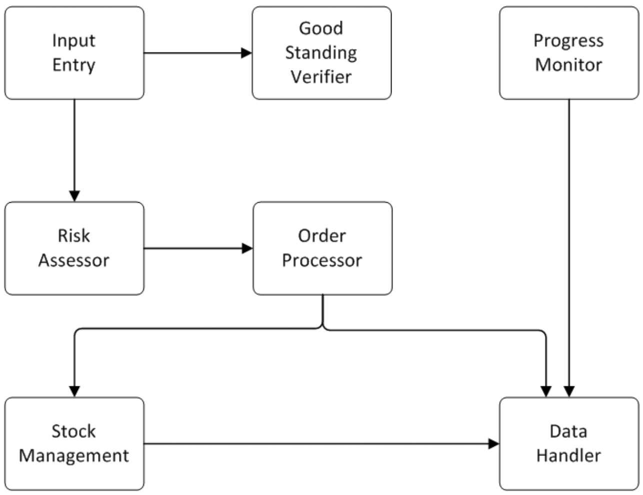


Fig. 3. Architecture diagram of the order system.

components of the architecture are presented as boxes and interactions between the components are represented by arrows. The customer request is received by the Input Entry component. Once the order is received, the Input Entry component passes appropriate data to the Good Standing Verifier component. Based on the results returned by the latter component, the Input Entry component asks the Risk Assessor component to evaluate the risk for accepting the order for certain kinds of customers. In case the risk is low, the Risk Assessor kicks of the proper processing the order by using the Order Processor component. The latter component makes use of the Data Handler component for dealing with the persistence aspects of the actual order. In parallel, the Order Processor component instructs the Stock Management component to deal with all stock related aspect of the order. The Stock Management component too makes use of the Data Handler component for persistency aspects. The Progress Monitor component allows monitoring the progress of the order at any time; for that purpose, this component makes use of the status information about the order available via the Data Handler component. It is important that all components that should be subject of the movement to a cloud environment are modeled explicitly. This is the case for both technical and business components.

Within the metamodel the "Architecture Model" part shown in Fig. 2 supports specifying the corresponding model. For example, the Input Entry component is an instance of `Component` with the `Name` attribute set to *Input Entry*. The Risk Assessor is an instance of `Component` with `Name` *Risk Assessor*. The arrow between these two components is realized by an instance of `Component Relation` with `Type` set to *InputEntryusesRiskAssessor*. The *Input Entry* component is `source of` the *InputEntryusesRiskAssessor* `Component Relation` and the *Risk Assessor* component is `target of` the *InputEntryusesRiskAssessor* `Component Relation`.

2.4. *Example — deployment level*

The various components of the architecture of the application are realized based on different technologies: The Input Entry component, the Good Standing Verifier component, and the Progress Monitor component are implemented as servlets in a corresponding Web server. The Risk Assessor component and the Order Processor component are realized as session beans in a JEE application server. Both, the Data Handler component as well as the Stock Management component are built as stored procedures directly within a database management system. Figure 4 exemplarily shows the corresponding deployment of the application. The components are also annotated by properties (depicted as rectangles) and variability points (depicted as "bowls") required being set during deployment in order to support the proper interactions between the components. These annotations are discussed in the next section.

Within the metamodel the "Deployment Topology" part shown in Fig. 2 supports the specification of the corresponding model. For example, the DBMS is represented as an instance of `Component` with Type attribute set to *DBMS*. It

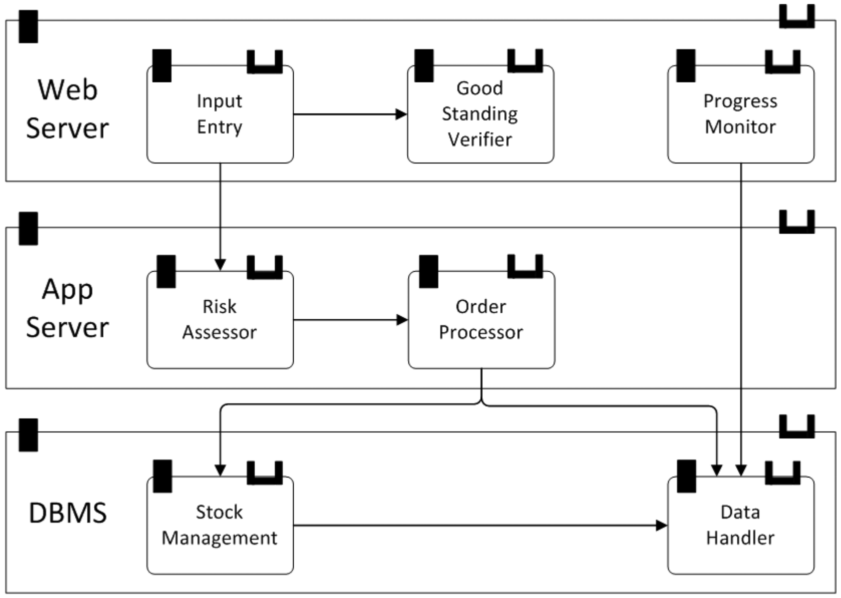


Fig. 4. Sample deployment of the order application.

is connected by an instance of the `contains` relationship with an instance of `Component` with `Name` set to *Stock Management* and `Type` set to *Database*. This component, for example, also has two `visible properties` with `Name` set to *Username* and *Password* and a `Value` set to the corresponding values as well as a `Variability Point` with `Name` set to *DatabaseName* and a `freeAlternative` where the database name can be set.

2.5. *Example — deployment parameterization and automatic installation*

In our sample application, we aggregate the middleware components into another component called *MWStack* to clearly distinguish middleware aspects and application aspects of the architecture of the sample application. Figure 5 shows how this aggregation is realized by an instance of `Component` called *MWStack*. This component `contains` two other components, a component of `Type` *AppServer* with `Name` *WebSphere*, and a component of `Type` *DBMS* with `Name` *DB*2.

Next, the "Automatic Installation" part as well as the "Deployment Parameterization" part of the metamodel from Fig. 2 is used to specify further deployment information beyond the pure middleware containment information. As shown in Fig. 5, the *WebSphere* component is `realized` by an `Implementation` of `Type` *OFV*. It `consists of` a `BLOB Artifact` that points to an element called *MWStack/Websphere.ovf* within the OVF file. This is achieved via its `FileRef` attribute. The artifact further `has` a `Variability Point` with `Name` *hostname*.

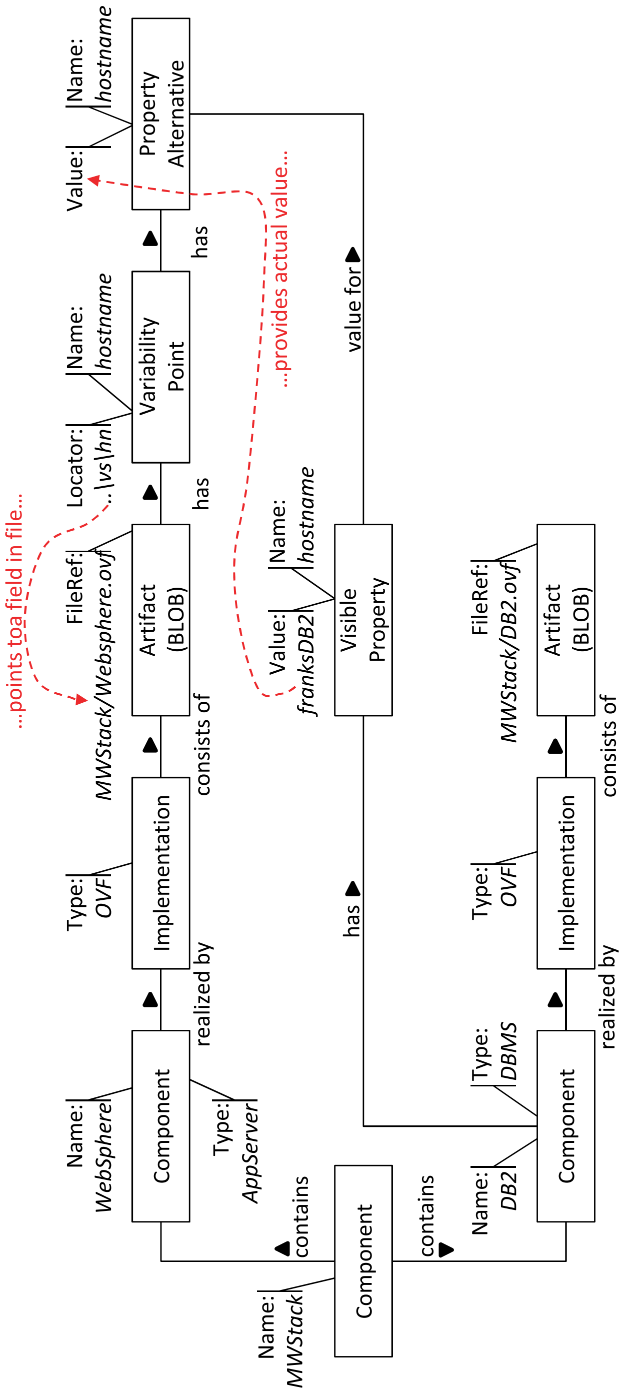
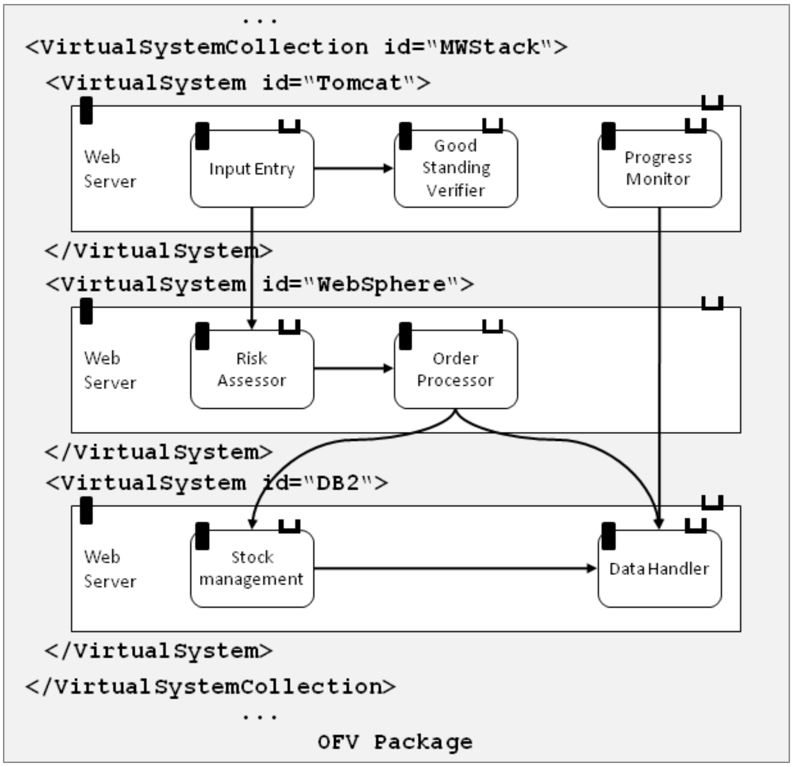


Fig. 5. Sample composite CAR component realized by virtual images.

The Locator attribute of this `Variability Point` points to the "…\vs\hn" element of the *MWStack/Websphere.ovf* file. A `Property Alternative` is defined for the *hostname*, i.e. the actual value of the *hostname* will be provided via a `Visible Property`. The corresponding `Visible Property` with `Name` *hostname* and `Value` *franksDB2* has been defined for the *DB2* component. This component is `realized by` an `Implementation` of `Type` *OVF* too, and this `Implementation` also `consists of` a `BLOB Artifact` with the `FileRef` attribute set to *MWStack/DB2.ovf*.

As a net effect, the value *franksDB2* of the *hostname* `Visible Property` of the *DB2* `Component` becomes the value of the *hostname* `Property Alternative` of the *WebSphere* `Component` which represents the hostname of the database system to be used by the *WebSphere* `Component`. At runtime this enables a connection of *WebSphere* to the corresponding *DB2*.

Figure 6 depicts the overall middleware stack required by the application as (a fragment of) an OVF file.[9] The `VirtualSystemCollection` element of the OVF file consists of three `VirtualSystem` elements each of which represents the virtual machine configuration of the particular piece of middleware. The figure is an overlay of the deployment model in Fig. 4 and the concrete syntax of an OVF file to show how individual components might point to corresponding OVF Virtual Systems.



```
...
<VirtualSystemCollection id="MWStack">
  <VirtualSystem id="Tomcat">
  </VirtualSystem>
  <VirtualSystem id="WebSphere">
  </VirtualSystem>
  <VirtualSystem id="DB2">
  </VirtualSystem>
</VirtualSystemCollection>
...
```

Fig. 6. Sample OVF overlay of the sample application.

However, the OVF file does not contain the model, but the model may point to the OVF artifacts. It graphically depicts that the `VirtualSystem` *Tomcat* hosts the application components Input Entry, Good Standing Verifier and Progress Monitor. The `VirtualSystem` *WebSphere* hosts the Risk Assessor and the Order Processor component. Finally, the `VirtualSystem` *DB*2 hosts the Stock Management and Data Handler Component. The fact that the *WebSphere* component is provided within an OVF file in the `VirtualSystem` *WebSphere* has been specified by means of the metamodel as sketched before.

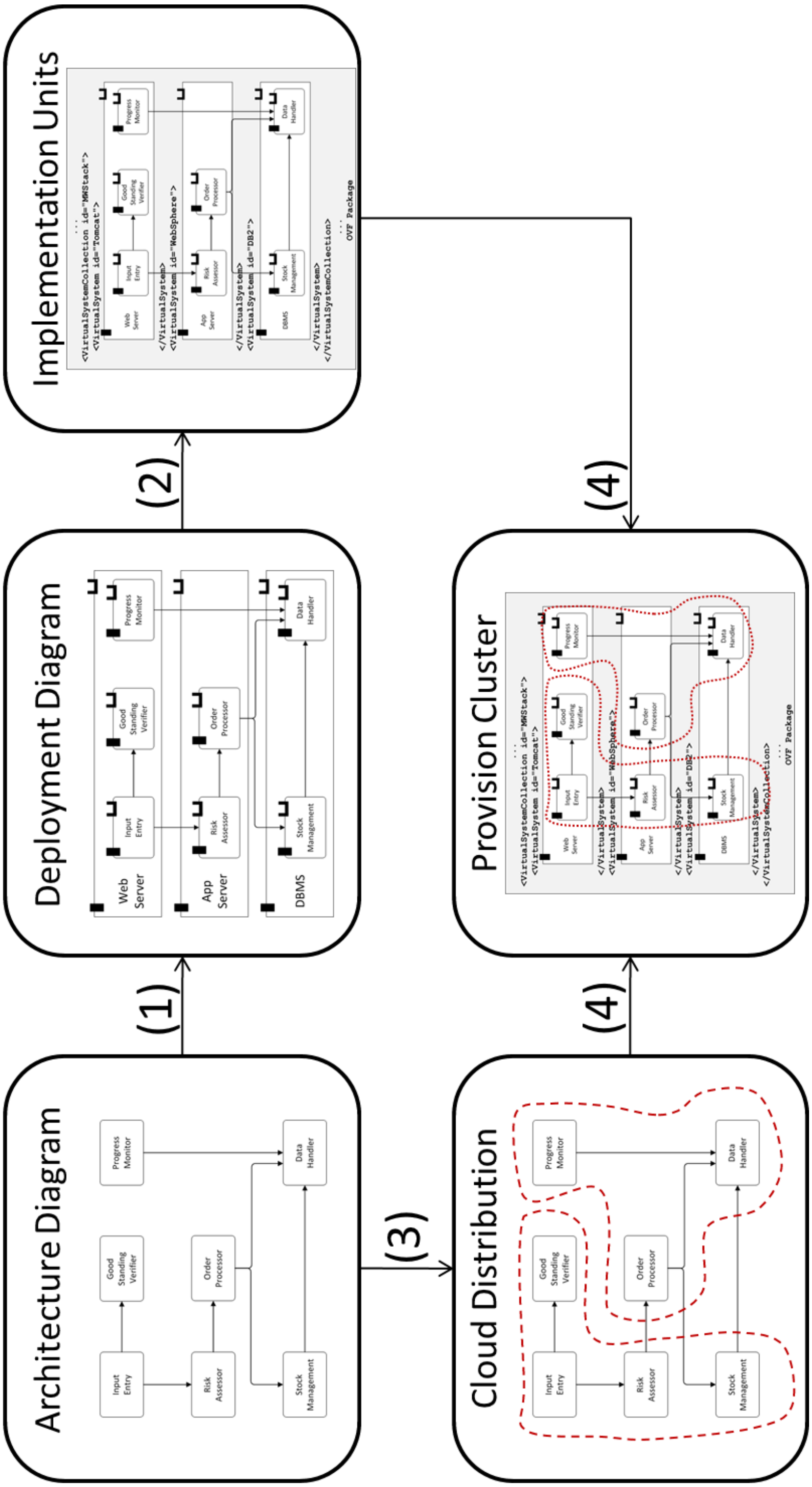
2.6. *MOCCA method details*

The main idea behind the method proposed is that an architecture model of the application is enriched by additional information and that this enriched model becomes the basis for automatically rearranging the application and provisioning the rearranged application in different clouds. Figure 7 shows the major artifacts created by following the MOCCA method.

One kind of additional information represents deployment information: the architecture model is combined with a deployment model of the application, and deployment relevant parameters are added. The other kind of additional information is about implementation units such as virtual images of the application that are associated with the components of the combined model. Finally, additional information may specify data associated with the architectural components and the interactions in-between these architectural components, and this data can be used to decide on an optimal rearrangement of the application (see Sec. 3.2).

The enriched architecture model is the basis for determining which part of the application is moved to which cloud, i.e. it is the basis for determining how the application should be rearranged. The rearrangement of the architectural components into groups of components is referred to as a *cloud distribution*; Fig. 7 indicates a cloud distribution in its lower left part. A cloud distribution is a disjoint partition of the set of all architectural components of the application into groups that are built according to the "cohesiveness" of the components. Cohesiveness is decided based on the third kind of additional information mentioned before that may be added to the architecture model and determines whether components have to be provisioned in the same cloud (see Sec. 3.1).

After deriving the cloud distribution for an application, the implementation units associated with each component of a group within the cloud distribution are bundled with the corresponding group of components. This result is called a *provision cluster*; Fig. 7 indicates a provision cluster in its lower part. Each such bundle of a provision cluster can be automatically provisioned (in a different cloud). As the result of provisioning, the overall collection of provisioned bundles is set up based on the deployment relevant parameters captured before such that the rearranged application is operable again.



```
...
<VirtualSystemCollection id="MWStack">
 <VirtualSystem id="Tomcat">
 </VirtualSystem>
 <VirtualSystem id="WebSphere">
 </VirtualSystem>
 <VirtualSystem id="DB2">
 </VirtualSystem>
</VirtualSystemCollection>
...
```

Fig. 7. Major artifacts of the MOCCA method.

Note that an implementation unit that is (part of) a virtual machine may have to be split or copied in the course of building a provision cluster because it may be associated with different components assigned to different bundles in a cloud distribution. For example, in Fig. 6 the fragment of the OVF file shown contains the `VirtualSystem` with identifier *WebSphere*. The two architectural components Risk Assessor and Order Processor are hosted by *WebSphere* as shown by the overlay of the architecture model and the OVF file in Fig. 6. Assume that Risk Assessor and Order Processor are decided to be moved to different clouds, i.e. they are assigned to different groups in the cloud distribution derived (as indicated in the box called "Provision Cluster" in Fig. 7). Since both, Risk Assessor as well as Order Processor will still require to be hosted by *WebSphere* after being moved to different clouds, the corresponding virtual machine has to be copied and bundled with the corresponding group in the resulting provision cluster. Section 3.1 and especially Definition 4 defines this precisely.

In a nutshell, the MOCCA method consists of the following major steps producing and combining artifacts resulting in a rearrangement and provisioning of an application in the cloud (see Fig. 7):

(i) As the basis, an architecture model of the application to be moved to the cloud has to be provided.
(ii) Furthermore, a deployment model of the application is required. Transition (1) in Fig. 7 represents the enrichment of the architecture model with deployment information.
(iii) Also, the architecture model is rearranged into groups of components that belong into the same cloud. Figure 7 depicts this as transition (3) that creates a cloud distribution from the architecture model. Note, that the creation of the cloud distribution and the deployment model can be performed in any order, even in parallel.
(iv) To support automatic provisioning, all implementation units must be provided that are required to actually run the application. Transition (2) in Fig. 7 indicates that this implementation information is added to the combined architecture/deployment model.
(v) Finally, the cloud distribution and the combined architecture/deployment model annotated with the required implementation units are combined into a provision cluster: the joint transition (4) in Fig. 7 represents this step. The provision cluster represents all the information needed to provision the rearranged application into its target clouds.

The creation of these artifacts can be supported by corresponding tools: in Sec. 4 we present the architecture of a corresponding tool suite and describe the individual tools of this suite; the appendix shows screenshots of the implementation of these tools. But it should be explicitly noted that the MOCCA method itself is independent of any specific tool: it provides a procedure of steps to be done and artifacts to create in order to move an application to the cloud. The artifacts could be created by any tool: for example, the architecture model could be drawn by pencil on a sheet of paper, could come as a set of power point slides, could be modeled via the ACME tool,[10] the Cafe tool[7] or the VBMF tool,[11] as a UML model and so on. But the tool suite presented in Sec. 4 supports the proposed method seamlessly. In the prototypical experiments performed in practice, all steps of the MOCCA method have been executed.

Figure 8 shows the procedural details of the MOCCA method as a BPMN[12] process model. The process begins with a task that provides an architecture model. This architecture model might already exist and is simply retrieved, or it is explicitly created by this task. Next, the cloud distribution of the application has to be determined and deployment information is to be provided: the process model represents these activities as expanded subprocesses with corresponding names. These two subprocesses may be performed in parallel or in any order.

The Determine Cloud Distribution subprocess begins with a decision whether or not the cloud distribution is derived by manually partitioning the architectural components of the architecture model or not. If a manual partitioning is performed the Provide Cloud Distribution task outputs the cloud distribution. If an automatic partitioning is chosen, the architecture model must be labeled by appropriate information within the corresponding task shown. Once the labels have been provided, the cloud distribution is automatically computed by the following task (Secs. 3.2 and 3.3 detail how this is achieved). The usage of MOCCA in practice was based on manual partitioning because the practitioners have been skeptical about automatic partitioning; nevertheless, the manual distribution chosen could be confirmed by the automatic partitioning afterwards.

The Provide Deployment Information subprocess starts with a task that provides the deployment model of the application; again, this model might already exist and is simply retrieved by the task, or the deployment model is created by that task. If some of the artifacts that represent implementation units are (part of) virtual images, these virtual images are provided in a separate task. As mentioned before, the practical usages exploited Provider Supplied types of `Implementations` of middleware components, i.e. the task Provide Virtual Images has not been performed. In any case, the task Define Implementation Artifacts associated the architectural components as well as the deployment components with their implementation units; especially, components whose implementation is provided as virtual images are linked to the corresponding `VirtualSystem` elements in OVF files (assuming OVF as format). Finally, the deployment relevant parameters are defined.

Once the cloud distribution as well as the deployment information is available, the implied provision cluster is automatically computed. Based on this information, the appropriate provision flows are automatically generated (see Sec. 4.3). Finally, the provisioning flow is executed resulting in the installation and proper deployment of the rearranged application in the cloud.

As indicated before, the method we propose can be used in a whole spectrum of scenarios each of which relate to a different degree of automation for moving

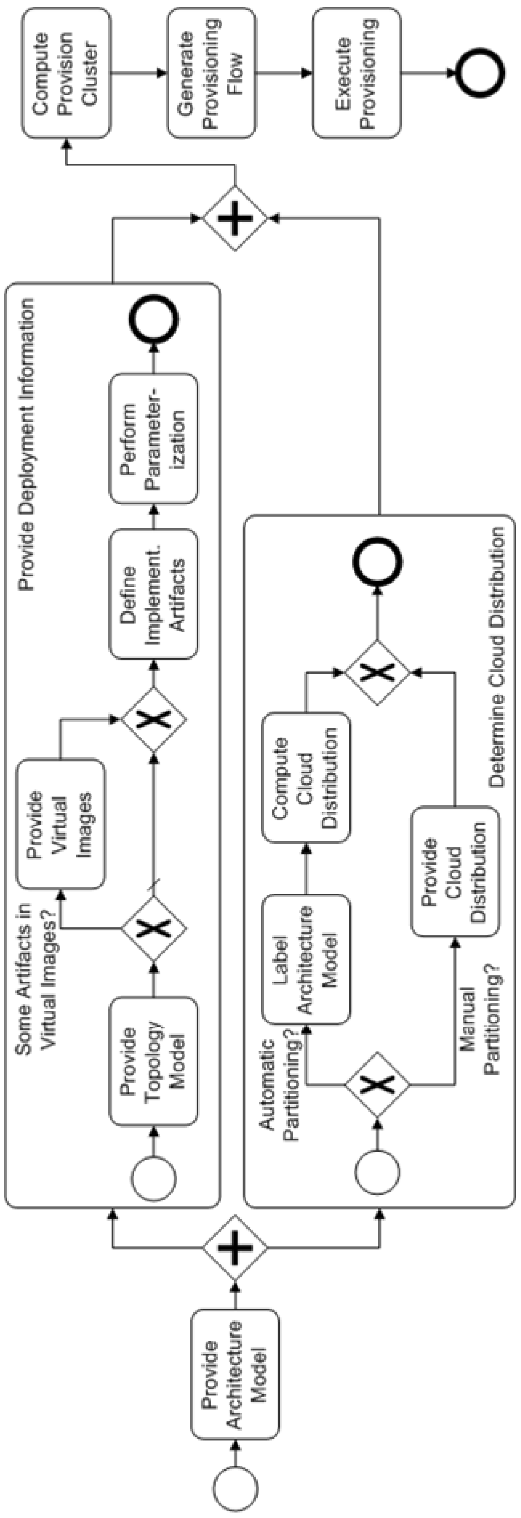

Provide Architecture Model
Provide Deployment Information
Provide Topology Model
Some Artifacts in Virtual Images?
Provide Virtual Images
Define Implement. Artifacts
Perform Parameter-ization
Determine Cloud Distribution
Automatic Partitioning?
Label Architecture Model
Compute Cloud Distribution
Manual Partitioning?
Provide Cloud Distribution
Compute Provision Cluster
Generate Provisioning Flow
Execute Provisioning

Fig. 8. Process model representation of the method.

an application to the cloud: it is possible to move an application to the cloud without any tool support at all, or by supporting some of the steps of the method by tools, or by using an environment that supports all of the steps of the method by corresponding tools.

At the low end of the spectrum our method can be used without any tool support, i.e. it is then considered as a guideline for the major steps to be performed when moving an application to the cloud. In this case, all of these steps have to be performed manually relying completely on the skills and knowledge of human beings performing these steps. At the high end of the spectrum an environment build by a tool suite on top of Cafe is used (see Sec. 4), i.e. the major steps of our method are supported by the environment guiding users through these steps. Some of these steps require user input and while other steps will be executed by the environment in an automatic manner. The practical work of the authors has been at this high end of the spectrum, i.e. it has been supported by tools.

Furthermore, the granularity of the architecture model and of the virtual image provided significantly influences the flexibility of spreading the application across different clouds as well as the reuse of componentry across applications moved to clouds (see Sec. 2.7 for a more detailed discussion). Similarly, if the virtual image consists of the collection of images of the individual middleware elements without any of the proper application components to be deployed into and hosted by these middleware elements, the same middleware elements can easily become containers for components of different applications (see Sec. 2.8 for a more detailed discussion).

2.7. *Impact of application architecture model granularity*

Obviously, the finer the granularity specified in the architecture model (i.e. the more components are specified) the more possibilities to split and scatter the application exist. More components typically means to have more detailed and more specific metrical information about the interaction between the components, which in turn typically results in more optimization options and better optimization results in splitting the application (see Sec. 3.2).

Coarse grained models such as ACME models[10] tend to capture the high-level logical components ("building blocks") of an application. However, in order to automatically split and especially provide an application later on, more fine grained models such as the ones employed in Cafe[7] are needed. These fine grained models go deeper than modeling the high-level logical components of an application and their relationships by capturing also technical components relevant for distributing and hosting the application ("deployment architecture").[13] Such deployment architecture models add deployment-relevant components and cross-component configuration needs.

Deployment-relevant components are components that explicitly specify their deployment needs. The advantage of deployment-relevant components is that they often can be automatically deployed, while logical components typically require manual intervention because their opaque, not explicitly modeled different parts must be deployed on different middleware stacks. When explicitly modeling the deployment-relevant components and their dependencies (i.e. component X must be configured with the IP address of component Y) the provisioning infrastructure can then interpret the respective model when provisioning the application which is not possible for the coarse grained models.

Thus, refinement of logical components is advantageous. When being refined, logical components are typically split into multiple deployment-relevant components each of which is separately represented in the refined model. For example, a *Web Portal* logical component of a coarse grained architecture model might consist of both, a *Portal Engine* as deployment-relevant component that must be deployed on an application server, as well as a *Portal Database* deployment-relevant component that must be deployed on a DBMS. Specifying these two deployment relevant components explicitly in the architecture model allows to automatically provide and deploy them.

2.8. *Impact of virtual image content*

The content of the virtual images that get overlaid has impact on the quality of the installation, potentiality of security threats, licensing issues etc. For example, if the virtual image of an application server contains EJBs that are not needed by the application to be moved to the clouds, the installation will be polluted. Components that are not required for an installation may open up security holes. Finally, components that are not needed by an installation may require unnecessary payments of license fees. Obviously, superfluous components generate management efforts because corresponding management processes (ITIL processes) automatically take care of them, for example.

Thus, the balance is between a set of pre-defined “empty” virtual images and application-specific images containing both the middleware and the application components. The advantage of employing images that contain only the middleware stack is their reusability. Instances of such images can be reused across different applications that have the same middleware requirements without being burdened by superfluous components that are not needed in that particular application. In addition to that, one instance of such an image can be reused in multiple applications and thus the amount of instances of such images can be greatly reduced. Furthermore management and maintenance overhead of the images can be reduced if only a predefined set of virtual images can be used in applications.

However, limiting the amount of usable images to a set of predefined middleware images also imposes a set of challenges: To ensure usability of the predefined images in multiple scenarios the middleware images must be highly configurable which again makes their definition and use very cumbersome as a lot of configuration options must be defined and bound before they can be used. As a consequence, not all possible configuration options can be captured in a configuration model for these images. Thus, application components that can be deployed on top of these images must be able to live with the possible configuration options. This may be a viable option for application components that are developed with these restrictions in mind. However, when moving existing legacy applications into the cloud these may be "by chance" compliant to one of the possible configurations but may also not be compliant. In addition to that, when using predefined virtual images the corresponding provisioning infrastructure must be able to deploy application components on top of these virtual images. In case of virtual images that contain both the middleware components and the application components representing the complete application, this is not necessary.

To capture the advantages of both worlds, Cafe[7] employs an approach where pre-defined virtual images can be reused across multiple customers and applications. Additionally, the Cafe application metamodel allows the inclusion of custom virtual images that may have special combinations of middleware and application components that cannot be decoupled into a predefined image and an application component.

3. Formal Aspects

In this section, we describe some formal aspect of the MOCCA method. First, we provide a formal model of provision clusters (see Sec. 3.1). Next, we describe the derivation of cloud distributions and provision clusters as an optimization problem (*Cloud Distribution Problem*) in Sec. 3.2. Finally, in Sec. 3.3, we give an example for such an optimization problem and sketch a tool for automatically solving the cloud distribution problem.

3.1. *Provision clusters*

The core concept underlying the MOCCA method is that of a provision cluster (see Definition 4). To prepare its formal definition we need to define formally what cloud distributions (see Definition 1) and middleware deployments (see Definition 2) are.

Informally, a cloud distribution is a partitioning of the architectural components of an application (see bottom left model of Fig. 7). The partitions are determined based on some criteria ("labels" in Definition 6) that allow evaluating the cohesiveness of the corresponding components. For example, business logic components very frequently accessing a particular database handler component and exchanging lots of data with the database handler might be put into a single joint partition together with the database handler to minimize latency and data transfer cost by putting the whole partition in the same cloud or even onto the same machine. A set of such placements considering also the middleware required by the partitioned architectural components is called a provision cluster (see Definition 4). For example, the business components above require an application server and the mentioned database handler component requires a database system, i.e. the corresponding partition of the components of the provision cluster includes an application server and a database system (see bottom center model of Fig. 7).

Definition 1. (a) Let $\mathcal{A}$ be an application and $\mathcal{C}(\mathcal{A}) = \{C_1, \ldots, C_n\}$ be the set of architectural components of $\mathcal{A}$. A disjoint partition $D = \{P_1, \ldots, P_m\} \subseteq \wp(\mathcal{C}(\mathcal{A}))$ of $\mathcal{C}(\mathcal{A})$ is called a *cloud distribution* of $\mathcal{A}$ (where $\wp(M)$ denotes the powerset of a set M).

(b) A cloud distribution D is derived based on a set of criteria that are represented by a function Δ that evaluate the cohesiveness of elements of $\mathcal{C}(\mathcal{A}) = \{C_1, \ldots, C_n\}$ with respect to having to belong to a joint single cloud. When this is important to emphasize the cloud distribution is denoted as $D = \Delta(\mathcal{C}(\mathcal{A}))$.

Δ may cover a large spectrum of types of criteria reaching from "gut feeling" over "best practices" to the use of algorithms. For example, an architect may simply "know" based on experience which components must be put into one and the same cloud. Another option may be the use of patterns for determining which components must be placed jointly into a single cloud. Also, optimization algorithms for determining the best placement of each component can be used based on metrical information associated with each component (see Sec. 3.2); in this case, $\Delta = (\Omega, \Phi, \Psi)$ is a triple consisting of labels Ω, node-labeling map Φ, and edge-labeling map Ψ (see Definition 6).

Definition 2. Let $\mathcal{A}$ be an application and $\mathcal{M}(\mathcal{A}) = \{M_1, \ldots, M_r\}$ be the set of middleware components hosting at least one of the architectural components $C_j \in \mathcal{C}(\mathcal{A})$ of $\mathcal{A}$. $\mathcal{M}(\mathcal{A})$ is perceived as a disjoint partition $\mathcal{M}(\mathcal{A}) \subseteq \wp(\mathcal{C}(\mathcal{A}))$ by defining $M_i := \{C_j | C_j \in \mathcal{C}(\mathcal{A}) \wedge C_j \text{ is hosted by } M_i\}$ for $1 \leq i \leq r$. $\mathcal{M}(\mathcal{A})$ is called *middleware deployment* of $\mathcal{A}$. $M_i \in \mathcal{M}(\mathcal{A})$ is called the *container* of the architectural components of $\mathcal{A}$ it hosts.

Let the components $\mathcal{C}(\mathcal{A})$ of an application $\mathcal{A}$ be rearranged into the cloud distribution $\Delta(\mathcal{C}(\mathcal{A}))$ based on the criteria Δ. In general, components C_i and C_j originally belonging to the same container M_k will be assigned during the rearrangement to different P_s and P_t of the partition $\Delta(\mathcal{C}(\mathcal{A}))$, i.e. $C_i \in P_s \cap M_k$ and $C_j \in P_t \cap M_k$. The meaning of $M_k \cap P_s \neq \emptyset$ and $M_k \cap P_t \neq \emptyset$ is that some components of P_s as well as some components of P_t require after the rearrangement still to be hosted by a container "of the same kind" M_k.

To crisply define the situation we introduce the following set operation:

Definition 3. Let T, $T' \subseteq \wp(M)$ be two non-empty sets of subsets of the set M. The *deep intersection* of T and T' is defined as $T \Cap T' := \{t \cap t' | t \in T \wedge t' \in T'\} - \emptyset$.

I.e. the deep intersection "$\Cap$" of two sets of sets T and T' is not the intersection of the two sets themselves but it is the set of pairwise intersections of sets contained in the encompassing sets T and T'. Based on this definition, the set of middleware components required to host the rearranged set of components of an application $\mathcal{A}$ is the deep intersection of the middleware deployment and the cloud distribution of $\mathcal{A}$:

$$\mathcal{M}(\mathcal{A}) \Cap \Delta(\mathcal{C}(\mathcal{A})) = \{M_i \cap P_i | 1 \leq i \leq r \wedge 1 \leq j \leq m\} - \emptyset. \quad (1)$$

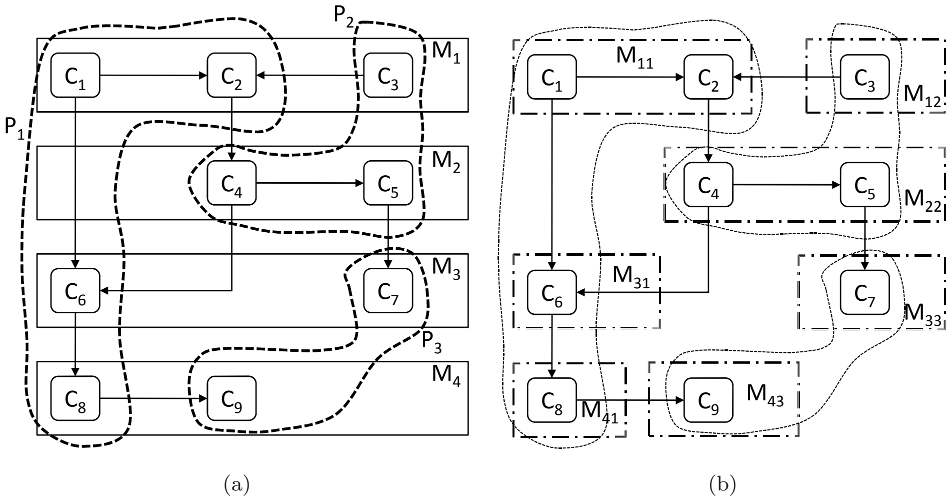


Fig. 9. Sample (a) Cloud distribution and (b) Provision cluster.

Figure 9 (a) shows a sample cloud distribution of an application. The underlying deployment topology shows which component C_i within the architecture model of the application is contained in (a.k.a hosted by) which middleware container M_k. The partitions P_s of the cloud distribution are represented by components C_i surrounded by dashed lines and containers of the middleware deployment are represented by rectangles. Intuitively, the partitions of the cloud distribution "tear apart" the containers, e.g. M_1 is split into M_{11} and M_{12}. Thus, the rearranged application requires two copies M_{11} and M_{12} (drawn as rectangles with dashed-bulleted lines in part (b) of Fig. 9) of the original middleware container M_1. While M_1 originally hosted C_1, C_2, and C_3, after rearrangement M_{11} will host C_1 and C_2, and M_{12} will host C_3. Part (b) of the figure groups those application components that can be provisioned into separate clouds together with the middleware containers required to host the corresponding components by lines with narrow dashes.

The following definition introduces this formally:

Definition 4. Let $\mathcal{A}$ be an application, $\mathcal{M}(\mathcal{A}) = \{M_1, \ldots, M_r\}$ be the set of containers hosting at least one architectural component of $\mathcal{A}$ and $\Delta(\mathcal{C}(\mathcal{A})) = \{P_1, \ldots, P_k\}$ be a cloud distribution of $\mathcal{A}$. Then, the deep intersection of $\mathcal{M}(\mathcal{A})$ and $\Delta(\mathcal{C}(\mathcal{A}))$, i.e.

$$\mathcal{M}(\mathcal{A}) \Cap \Delta(\mathcal{C}(\mathcal{A})) = \{M_i \cap P_j | 1 \leq i \leq r \text{ and } 1 \leq j \leq k\} - \emptyset \tag{2}$$

is the set of containers required to host the components of the rearranged application $\mathcal{A}$. The pair $\Pi(\mathcal{A}) := (\Delta(\mathcal{C}(\mathcal{A})), \mathcal{M}(\mathcal{A}) \Cap \Delta(\mathcal{C}(\mathcal{A})))$ is called a *provision cluster* of $\mathcal{A}$.

3.2. *Cloud distribution problem: Computing cloud distributions and provision clusters*

A cloud distribution of an application $\mathcal{A}$ can be computed by evaluating metrical annotations (or "labels") of the architecture model of $\mathcal{A}$. This implies that a provision cluster of $\mathcal{A}$ can be automatically proposed by computing the deep intersection of the computed cloud distribution and the given middleware deployment of $\mathcal{A}$.

The problem of computing a cloud distribution is mapped to a combinatorial optimization problem, more precisely to a variant of the graph partitioning problem[14] in which partitions do not necessarily have similar size.

In order to formalize the problem of computing a cloud distribution of an application $\mathcal{A}$, the notion of an architecture model of $\mathcal{A}$ is defined as a directed graph the nodes of which are the components of $\mathcal{A}$ and the edges of which are the interactions between the components of $\mathcal{A}$.

Definition 5. Let $\mathcal{A}$ be an application, $\mathcal{C}(\mathcal{A}) = \{C_1, \ldots, C_n\}$ be the set of architectural components of $\mathcal{A}$ and $\mathcal{R}(\mathcal{A}) \subseteq \mathcal{C}(\mathcal{A}) \times \mathcal{C}(\mathcal{A})$ be the set of interactions between the architectural components of $\mathcal{A}$. The directed graph $\mathcal{G}(\mathcal{A}) = (\mathcal{C}(\mathcal{A}), \mathcal{R}(\mathcal{A}))$ is called the *architecture model* of $\mathcal{A}$.

To decide on the cohesiveness of architectural components of an architecture model, the model must be instrumented by different kinds of metrical information (collectively referred to as "labels"). Metrical information might be associated with the components of the architecture model (i.e. $\mathcal{C}(\mathcal{A})$: the nodes, architectural components or "boxes", respectively), with its interactions (i.e. $\mathcal{R}(\mathcal{A})$: the edges, interactions or "arrows", respectively) or even with both. For example, when the overall cost of hosting an application is to be evaluated, the cost of hosting each architectural component of the application will be assigned as label to the nodes of the architecture model. When the amount of data transferred across the network is to be evaluated, the amount of data exchanged between two components will be assigned as metrics to each edge of the architecture model.

The following definition introduces labeling of directed graphs formally:

Definition 6. Let $G = (N, E)$ be a directed graph and let $\Omega = \{\Omega_1, \ldots, \Omega_n\}$ be a set of non-empty sets, where each Ω_i is called set of *labels* of a certain type. A label $f \in \Omega_i$, $1 \leq i \leq n$, is a function $f : D_f \to \mathbb{R}$, where D_f denotes the domain of f. If a non-empty set $\Omega(N) \subseteq \Omega - \emptyset$ and a map

$$\Phi : N \to \bigtimes_{\omega \in \Omega(N)} \omega \tag{3}$$

has been defined, G_Φ is called a *node-labeled* graph, Φ is called *node-labeling map*, $\Omega(N)$ is the set of *node labels*. If a non-empty set $\Omega(E) \subseteq \Omega - \emptyset$ and a map

$$\Psi : E \to \bigtimes_{\omega \in \Omega(E)} \omega \tag{4}$$

has been defined, G_Ψ is called an *edge-labeled* graph, Ψ is called *edge-labeling map*, $\Omega(E)$ is the set of *edge labels*. $G_{\Phi,\Psi}$ is called a *labeled* graph if it is both, node-labeled as well as edge-labeled with node-labeling map Φ and edge-labeling map Ψ.

Architecture models are directed graphs and, thus, can be turned into (node-, edge-) labeled graphs by annotating the architectural components $\mathcal{C}(\mathcal{A})$ or the interactions $\mathcal{R}(\mathcal{A})$ of the model with corresponding metrical information (i.e. with labels). Examples for labels of architectural component are average response time of a component, the cost of hosting a component, or the trust sensitivity of a component. Examples of interaction labels are number of calls the source of the interactions performs on the target of the interaction, or the amount of data transferred between the components. Within the MOCCA metamodel (Fig. 1) labels are specified as instances of the `Label` entity type. An instance of `Label` is assigned to a `Component` (i.e. architectural component) or a `Component Relation` (i.e. interaction) via the corresponding `has` relationship type.

Note that a label f that is a constant function (i.e. $f(x) = c$ for all $x \in D_f$) is considered as fixed value c. For example, each interaction $r \in \mathcal{R}(\mathcal{A})$ can be associated with the average amount of data d_r transferred per hour across r as a label (that is fixed, i.e. that is a constant function), i.e. $\Psi(r) = d_r$. This example also shows that Ω_i is typically really a set with many elements: each interaction is likely associated with at different value d_r, and these values are of the same type "data transferred per hour" and are thus grouped into a corresponding set of labels $\Omega_{\text{data/hour}}$. Another example of a set of labels is the set of cost functions f_{provider} each of which returns the cost of hosting a piece of software at a certain provider. This cost is based on a set of parameters like size of the image of the software, number of invocations per hour making up the domain of f_{provider}, and these parameters may be different for different providers. Thus, a single cost function is not sufficient to determine the cost of hosting a partition of the components of an application at different providers, but a set $\Omega_{\text{costFunction}}$ of provider dependent cost functions is needed.

For each type of label Ω_i we assume a corresponding aggregation function $\alpha(\Omega_i)$ (or α_i for short) that is used to appropriately aggregate a set of label values of nodes $\{\pi_i(\Phi(m))|m \in M \subseteq N\}$ (where π_i is the projection of a tuple onto its i-th component) or label values of edges $\{\pi_i(\Psi(m))|m \in F \subseteq E\}$. If Ω_i is a set of node labels, $\alpha(\Omega_i)$ is a function

$$\alpha(\Omega_i) : \wp(N) \to \mathbb{R}; \tag{5}$$

if Ω_j is a set of edge labels, $\alpha(\Omega_j)$ is a function

$$\alpha(\Omega_j) : \wp(E) \to \mathbb{R}. \tag{6}$$

For example, if Ω_i represents the cost for hosting an architectural component $\mathcal{C}(\mathcal{A})$, the aggregation function $\alpha(\Omega_i)$ (or α_i for short) is simply the sum of all the hosting costs associated with all the architectural components within $M \subseteq N$:

$$\alpha(\Omega_i)(M) = \alpha_i(M) = \sum_{m \in M} \pi_i(\Phi(m)). \tag{7}$$

In general, each node and each edge of a labeled graph is associated with more than one label (the mapping of all labels to real values is here used due to simplification reasons and to focus on the MOCCA tool-chain). For example, an architectural component may be associated with the cost of its hosting and its trust sensitivity.

Typically, different types of labels have different importance (i.e. different priorities); for example, the trust level achieved when hosting a component with a certain provider may be more important than its low hosting cost offered. The different priorities of the different types of labels Ω_i is reflected by associating a particular *priority* $\varrho(\Omega_i) \in \mathbb{R}$ (or ϱ_i for short) with each particular Ω_i. However, sometimes users may not have clear what their priorities are. In the case that users cannot determine their exact priorities or that there are no priorities given by the user at all, they all can be set to the same value and the proposed method will continue to work anyways. We will further provide different priorities within the same type of label in future work.

When partitioning the nodes of a directed graph $G = (N, E)$ a corresponding partition of the edges of G can be defined in a canonical manner: all edges pointing to a particular set of nodes are grouped into one and the same set of edges. More precisely, for each disjoint partition of nodes $\{P_1, \ldots, P_m\} \subseteq \wp(N)$ the *induced edge partition* $\{Q_1, \ldots, Q_m\} \subseteq \wp(E)$ is defined via $Q_j := \{e \in E | \pi_2(e) \in P_j\}$, $1 \leq j \leq m$.

With this terminology the problem of computing a cloud distribution can be defined as follows.

Definition 7 (*Cloud Distribution Problem*). Let $\Omega = \{\Omega_1, \ldots, \Omega_n\}$ be a set of node labels, $\{\alpha(\Omega_1), \ldots, \alpha(\Omega_n)\}$ be corresponding aggregation functions, and $\{\varrho(\Omega_1), \ldots, \varrho(\Omega_n)\}$ be priorities of the labels. Furthermore, let $\mathcal{G}_{\Phi,\Psi}(\mathcal{A}) = (\mathcal{C}(\mathcal{A}), \mathcal{R}(\mathcal{A}))$ be a corresponding labeled architecture model with node-labeling map Φ and edge-labeling map Ψ. The *Cloud Distribution Problem* is to find a disjoint partition $\{P_1, \ldots, P_m\} \subseteq \wp(\mathcal{C}(\mathcal{A}))$ such that

$$\sum_{i=1}^{m} \left(\sum_{\omega \in \Omega(\mathcal{C}(\mathcal{A}))} \rho(\omega) \cdot \alpha(\omega)(P_i) + \sum_{\omega \in \Omega(\mathcal{R}(\mathcal{A}))} \rho(\omega) \cdot \alpha(\omega)(Q_i) \right) \tag{8}$$

becomes a minimum. Such a partition $\{P_1, \ldots, P_m\}$ is a cloud distribution of $\mathcal{G}_{\Phi,\Psi}(\mathcal{A})$.

In the formula above (which is referred to as *target function*), $\{Q_1, \ldots, Q_m\}$ denotes the edge partition induced by $\{P_1, \ldots, P_m\}$, $\Omega(\mathcal{C}(\mathcal{A})) \subseteq \Omega$ are the node labels of $\mathcal{G}_{\Phi,\Psi}(\mathcal{A})$, and $\Omega(\mathcal{R}(\mathcal{A})) \subseteq \Omega$ are the edge labels of $\mathcal{G}_{\Phi,\Psi}(\mathcal{A})$.

3.3. *Example and experiments: Solving the cloud distribution problem*

Next, we describe a solution of the cloud distribution problem by using simulated annealing[15] as well as a combination of multiple optimization methods. An implementation of our solution is provided as an adaptation of a tool that we introduced in Ref. 16 (for implementation details see there). To automatically compute an optimized cloud distribution we exploit (i) hillclimbing, (ii) simulated annealing, (iii) an evolutionary algorithm and, (iv) a hybrid approach containing elements from (i) and (ii). We extended the prototype from Ref. 16 with a new data structure covering both, architecture models as appropriate graphs and cloud structures; furthermore, the algorithms have been adapted to solve the cloud distribution problem.

We continue our running example and define labels and associate them with the components and component relations of the architecture diagram from Fig. 3 as shown in Fig. 10. Each label is defined as an instance of `Label` (see the metamodel in Fig. 1) with appropriate `Name` and `Value` attributes. For example, compute units labels are instances of `Label` having their `Name` attribute set to *ComputeUnits*; and a *ComputeUnits* label with actual value 2 is an instance of `Label` having its `Value` attribute set to 2. `Labels` are associated with `Components` and `Component Relations` by means of instantiating the appropriate `has` relationship. In Fig. 10, all components and component relations of the architecture diagram from Fig. 3 have been labeled.

Clouds are modeled as tuples of properties relevant for deciding the cloud distribution problem. Which properties to use, i.e. to decide which properties are appropriately characterizing the cloud candidates, are dependent on the specific situation. In our example, we characterize a cloud by four properties: (i) *computeUnitCosts* represents the amount of money charged for each compute unit, (ii) *cloudSecurityLevel* represents the security level a corresponding cloud can provide, (iii) *innerEdgeDataThroughputCosts* represents the amount of money to be paid per data unit transferred within the corresponding cloud, and (iv) *outerEdgeDataThroughputCosts* represents the amount of money to be paid per data unit out of and into the corresponding cloud. The concrete property values of two different clouds used as basis for our example are shown in Table 1.

Let Γ be the search space that represents a set of valid cloud distributions, $x, y \in \Gamma$ two valid cloud distributions and $U(x) = \Gamma \backslash \{x\}$ the environment of x; then the heuristic hillclimbing algorithm can be sketched as follows: the algorithm first selects a random element x from the given search space Γ and calculates the initial fitness of this element (see step 1 in Algorithm 1); the "fitness" of an element x is represented by the value of the target function (Definition 7) for this element. In our scenario, some of the labels associated with a component or a component relation are in fact formulas having as parameter the value of the label as well as one of the characteristic properties of the potential target clouds. For example, the label *ComputeUnits* represent the computing units consumed by a component, but its value must be multiplied with the corresponding cloud compute unit costs (*computeUnitCosts*) being a characteristic property of a particular cloud. Similarly, the label *dataThroughput* represent the data throughput of a component relation, but its value is multiplied with either *innerEdgeDataThroughputCosts* or *outerEdgeDataThroughputCosts* depending on where the communication partner is located. In our use data traffic within a single cloud is assumed to be free of charge while incoming and outgoing data traffic is with costs. This way, each term of the sum of the target function is evaluated and the total fitness value of a specific cloud distribution considering all labels is calculated as defined in Definition 7. At the end the cloud distribution with minimal fitness is selected as optimal solution (see steps 6 and 7 in Algorithm 1).

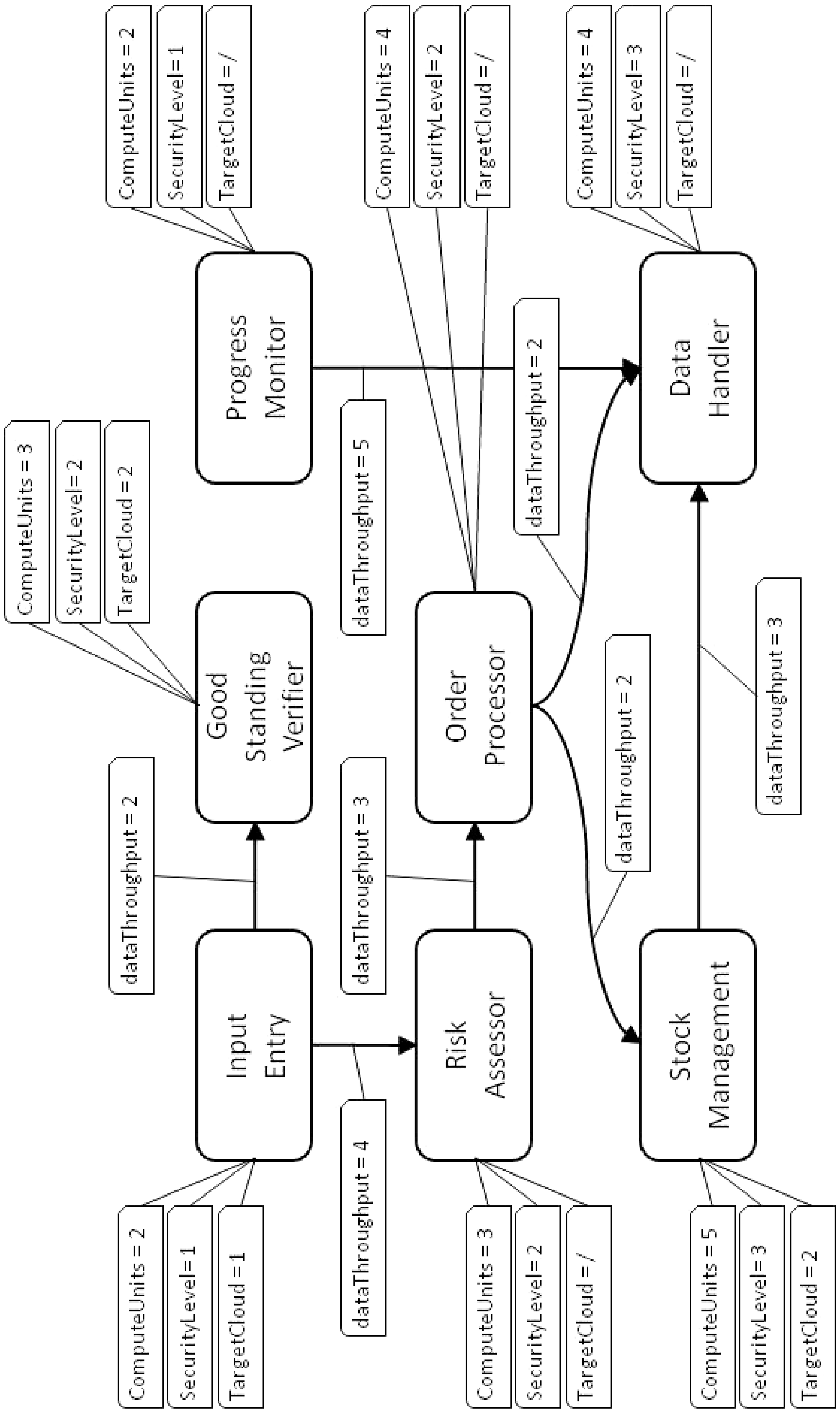


Fig. 10. Sample labeling of the architecture graph.

Table 1. Cloud definition.

Cloud property	Cloud 1	Cloud 2
computeUnitCosts	0.5	0.4
cloudSecurityLevel	5	5
innerEdgeDataThroughputCost	0.0	0.0
outerEdgeDataThroughputCost	0.2	0.5

Algorithm 1 Hillclimbing

```
1:  select x ∈ Γ and calculate fitness(x)
2:  i = 0
3:  while (i < Number of Steps) do
4:    select neighbor: y∈U(x)
5:      calculate fitness (y)
6:      if fitness (y) ≤ fitness (x)
7:        x = y
8:      end if
9:      i = i+1
10: end while
```

To determine a valid neighbor of x in $U(x)$ to compare the calculated fitness values to (see step 4 in Algorithm 1) we additionally introduced two evaluation constraints named *targetCloud* and *securityLevel* implemented as node labels (see Fig. 10). The first one verifies if a component had to be stored on a specific or arbitrary cloud by comparing the node label with the corresponding cloud definition, and the second one verifies if the components security level (*securityLevel*) is less or equal the clouds provided security level (*cloudSecurityLevel*). We decided to use a *securityLevel* range between 0 and 5 (0 indicates lowest and 5 highest security level) to calculate the aggregation function introduced in Definition 7. Furthermore, this allows users to manipulate the cloud distribution based on their know-how or legislative guidelines, for instance. All labels including their concrete values and cloud properties can be found in Fig. 10 and Table 1, respectively.

Table 2 summarizes the measurements by using the different optimization methods we used in our experiments for one setting of label values: all optimization methods resulted in the same cloud distribution with the same fitness. This computed cloud distribution is shown in Fig. 11.

Table 2. Algorithm execution times.

Algorithm	Execution time (in ms)	Parameter	Fitness (in monetary units)
Hillclimbing	296	Number of steps = 50	13,2
Simulated Annealing	767	Number of steps = 200	13,2
Hybrid	869	Number of steps = 50	13,2
Evolutionary	3224	Number of steps = 50	13,2

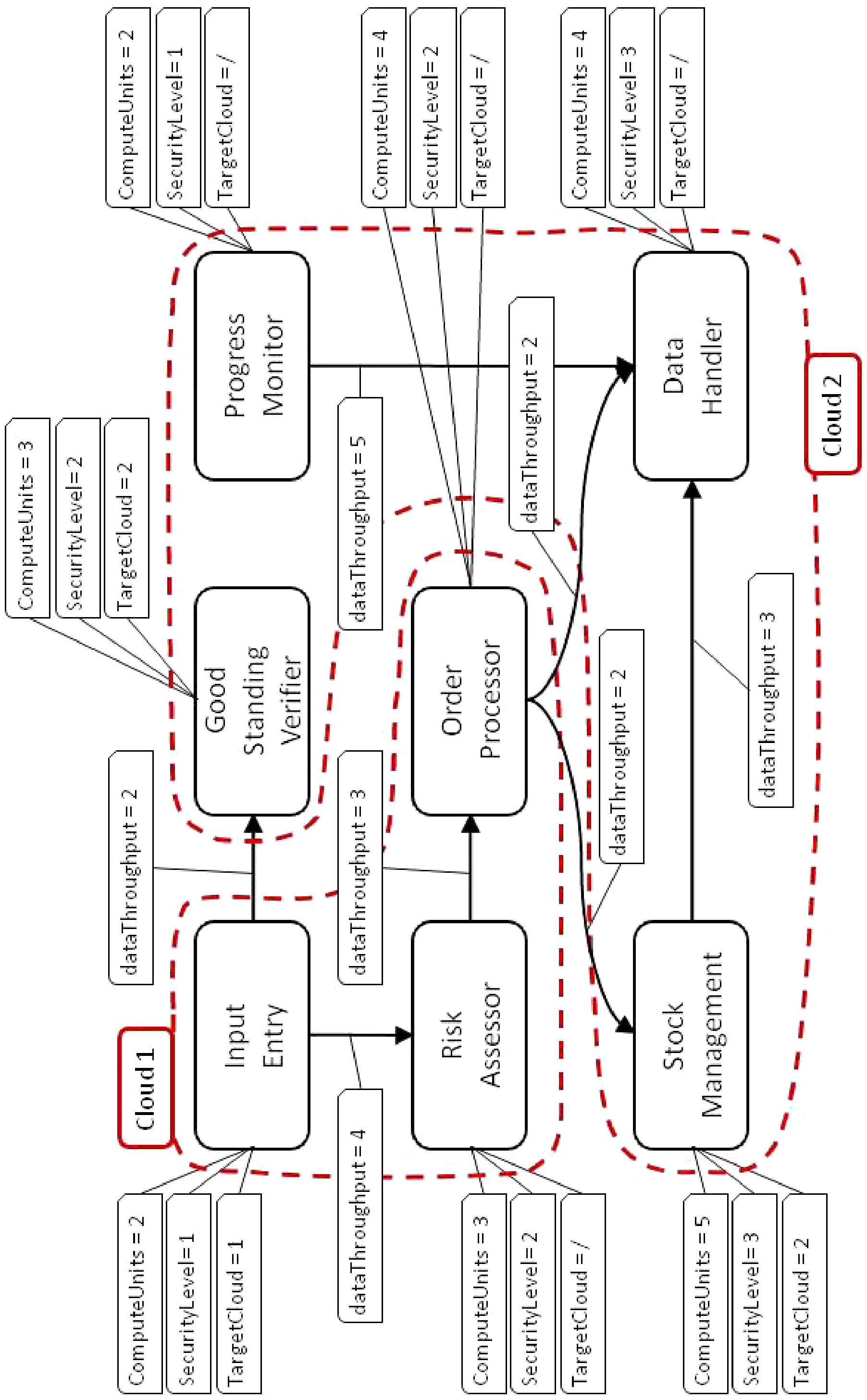


Fig. 11. Sample computed cloud distribution.

Additionally, we repeated the computations with various combinations of label values, and the different optimization methods always produced the same result. Thus, from a pure result perspective the concrete optimization method chosen was irrelevant in our experiments. But significant differences in execution time could be observed, which we ascribe to the various complexities of the different algorithms. Decreasing the number of steps each algorithm runs through up to a certain threshold the hillclimbing algorithm is performing even better than the others (i.e. it finds the optimal solution faster).

This tendency may change when increasing the number of architectural components or component relationships significantly, because of the well-known disadvantages of hillclimbing as local search algorithm. Then, the other algorithms may take the advantage of finding the global optimal solution instead of finding a single local optimal solution based on hillclimbing. Certainly the execution time will increase as well and maybe more calculation steps are necessary to find an optimal solution.

One project from practice rearranged an application based on trust aspects. These aspects correspond to *securityLevel* lables and *cloudSecurityLevel* labels above. Another project rearranged an application based on costs of hosting individual components of the application. These costs had been derived by licensing costs of the middleware components as well as corresponding hosting costs of the cloud provider. The cloud distributions had been specified manually by the practitioners, which could be reproduced algorithmically. If more labels and especially a mixture of labels of different kinds will be used (e.g. costs, times, availability, security etc.) it is expected that manually determined cloud distributions will often fails to be optimal and the automatically determined cloud distributions will be "better": But this has not be verified in practice yet.

4. The MOCCA Tool

In this section, we describe the architecture of a tool supporting the MOCCA method and concepts. Especially, the overall architecture is given and the individual components of the tool are described. Finally, the role and use of the Cafe environment[7] is sketched.

4.1. *Overall architecture*

Figure 12 shows the overall architecture of the MOCCA tool. The tool consists of several components supporting the various artifacts of the method proposed. Architecture models (Definition 5) are modeled using the Architecture Modeler. Deployment topologies and models, middleware deployments (Definition 2) as well as deployment relevant parameters and installation relevant artifacts are specified by means to the Deployment Modeler. The cloud distribution (Definition 1), i.e. the split of the application is derived via the Cloud Distributor. Based on the cloud distribution and the middleware deployment the Provision Preparation component

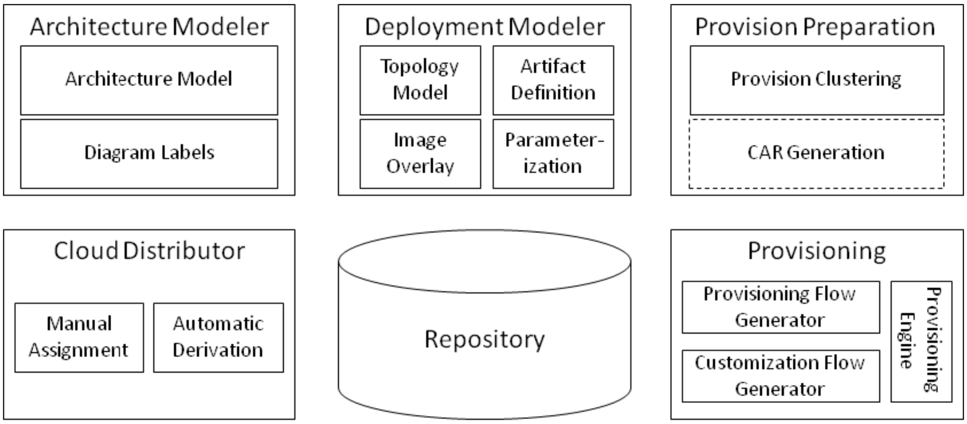


Fig. 12. Tool architecture.

determines the corresponding provision cluster (Definition 4). Together, this enables the Provisioning component to provision the split (i.e. rearranged) application in the various clouds.

The components exchange data via a shared repository. Basically, the schema of this repository is the metamodel described in Sec. 2.2. Note that the Provisioning component is basically the core of the Cafe environment. Since Cafe defines its own exchange format (Cafe ARchive, called CAR files) the Provision Preparation component supports the generation of CAR files in order to allow using Cafe implementations not sharing the same repository.

4.2. *Component descriptions*

The **Architecture Modeler** consists of two components, namely the Architecture Diagram component and the Diagram Labels component. The Architecture Diagram component supports the graphical modeling of architecture models, i.e. the architectural building blocks of an application as well as the relations between them; this component instantiates the "Architecture Model" part of the metamodel in Fig. 2. The Diagram Labels component is used to define properties (i.e. "labels" in Definition 6) relevant for deciding on the cohesiveness of architectural building blocks. Cohesiveness is decided based on properties of components or properties of interactions between components. Since interactions are represented by relations between architectural building blocks, these properties are associated with relations or with components of an architecture model as corresponding metrical information. Together, this results in instances of the "Optimized Clustering" part of the metamodel in Fig. 2. The use of the Diagram Labels component is required when cloud distributions should be proposed automatically by MOCCA.

The **Deployment Modeler** consists of the Deployment Diagram component that allows to graphically create the corresponding models, the Artifact Definition component, the Image Overlay component and the Parameterization component. The Deployment Diagram component supports the graphical modeling of the deployment topology of an application, i.e. it instantiates the "Deployment Topology" part of the metamodel in Fig. 2; especially, the *middleware deployment* of an application can be modeled. The Artifact Definition component allows specifying details about the implementation artifacts required to install a component in its runtime environment; this component instantiates the "Automatic Installation" part of the metamodel in Fig. 2. The artifacts needed are essentially the code files or packages (such as WAR files, or OVF images) that implement the component. These can be reused across different applications, for example the implementation of a Web service can be used in several applications. Depending on its implementation type, an artifact may be deployable on components of different types, i.e. a WAR file might be deployable on a component of type `Apache Tomcat`, or a component of type `JBOSS`. By defining artifacts representing virtual images (or parts thereof) as implementations of application components, the Image Overlay component supports overlaying the architecture of an application and virtual images. The Parameterization component allows defining both, deployment relevant properties and variability points of a component as well as the relations between them. Thus, this component is used to instantiate the "Deployment Parameterization" part of the metamodel in Fig. 2.

The **Cloud Distributor** is used to determine a *cloud distribution* for a given application. A cloud distribution can be defined manually by using the Manual Assignment component. If the Diagram Labels component has been used to annotate the architecture model of an application with cohesion relevant properties, the Automatic Derivation component can be used: it will automatically propose a cloud distribution based on the optimization algorithms described in Sec. 3.2.

The **Provision Preparation** component especially derives the *provision cluster* of an application based on the middleware deployment determined by using the Deployment Modeler and the cloud distribution determined by the Cloud Distributor. The corresponding deep intersection is computed by the Provision Clustering component of the Provision Preparation. As a result, the application template of the rearranged application is build. Furthermore, the "CAR Generation" component could be used to generate the CAR file for the rearranged application, i.e. the file format used by the Cafe tool and a proposed interchange format for composite cloud applications.

The **Provisioning** component is a subset of the Cafe environment as used in MOCCA; the corresponding functionality is described in Sec. 4.3 (see also Ref. 17). Basically, the Customization Flow Generator generates a customization workflow that derives the properties required for provisioning and deployment of the rearranged application. This is an important step in the provisioning process as the customization flow gathers the required values to bind variability points either from a user or from the associated visible properties and overwrites the configuration settings of the corresponding artifacts as indicated by the locators. For example, EPR values or JNDI properties of components are overwritten with concrete values obtained from the visible properties of other components. The customization workflow is used by the provisioning workflow generated by the Provisioning Flow component. The provisioning flow is enacted by the Provisioning Engine to finally install the rearranged application in the target cloud environments.

4.3. *Usage of Cafe for performing actual deployment*

Cafe maps OVF artifacts like virtual systems to separate components. Currently, Cafe assumes that a single OVF file represents a single component. Thus, in case a provisioning cluster contains an OVF file that contains the virtual image of more than one component (i.e. more than one virtual systems), this file must be split into separate OVF files manually. A straightforward extension of Cafe will either perform this split automatically (thus, using the existing Cafe unchanged) or will support OVF files with virtual images of multiple components.

From the deployment parameterization of an application (which is called "variability model of an application template" in Cafe) the Cafe infrastructure generates a so-called customization flow that deals with the binding of the variability points contained in the deployment parameterization. The provisioning flow is a workflow that represents the variability points, their alternatives, their enabling conditions and the dependencies between the variability points. The provisioning flow ensures a complete and correct customization of the application during deployment. "Complete and correct" means that (i) each variability point is bound and (ii) the rules imposed on the binding of variability points, i.e. which alternatives may be selected and in which order the variability points are bound, are followed. The generation of customization flows from variability models is described in detail in Ref. 17.

The deployment topology and the automatic implementation artifacts of an application (called "application model" in Cafe) as well as the dependencies between components induced by the deployment parameterization are interpreted by the Cafe provisioning environment in the following way: First a so-called "provisioning order graph" is generated. The provisioning order graph specifies in which order the components of the application must be provisioned. Three rules apply here: (i) before a component can be provisioned all components that transitively contain this component must be provisioned; (ii) before a component can be provisioned, all its variability points that must be bound at pre-provisioning time must be bound. Thus, all components to which a given component is connected via a property alternative whose associated visible properties become only available at runtime must be provisioned before the given component; (iii) components that do not have any dependencies on each other can be provisioned in parallel.

A provisioning flow can be generated from the provisioning order graph that performs the provisioning in the right order as follows: For each node in the graph so-called "provisioning activities" are added to the workflow model, these are connected via control connectors that represent the dependencies. This way the three rules above are ensured. The provisioning activities then contain activities to bind the pre-provisioning variability points of a component by calling the provisioning flow who will then either prompt the user for inputs if the deployment parameterization requires it or queries the already provisioned components for their respective visible properties. These are always already available as the ordering of the provisioning of the components follows rule (ii) above.

All already provisioned components in Cafe are represented by so-called "component flows" that provide a unified interface of components at different providers to the provisioning infrastructure. In order to deploy a component on an already deployed component, the deploy operation of the component flow of this component is called with the location of the repository in which the component that must be deployed is located. In case the component to be deployed is a virtual image (for example an OVF image) the component flow that represents the hypervisor of the provider that will later run the OVF image is called along with the repository location in which the customized OVF image is deployed. This operation is then mapped by the component flow to a hypervisor-specific operation that starts a new virtual image from a virtual image package such as OVF. When starting the new virtual image the hypervisor also starts the activation engine which starts the corresponding scripts contained in the virtual image. In case other components must be deployed on the infrastructure contained in the virtual machine, the component flow for the hypervisor starts a component flow that can deploy other components on the middleware component contained in the virtual machine. This component flow can be developed specifically for the virtual image or can be a standard component flow that makes use of the deployment interface of the component in the virtual machine, for example, a standard component flow could copy a Web application archive to a specific directory in the virtual machine or could deploy a process archive via the deployment Web service of the BPEL[18] engine contained in the virtual machine. Thus a component flow that implements the deploy operation for components that must be deployed on the middleware component in the virtual machine must be deployed in the Cafe environment before the corresponding application can be provisioned.

5. Case Study

In this section, we report about the actual move of the sample application introduced in Sec. 2.3 into the cloud. First, the implementation of the architecture components is described followed by their corresponding Deployment Parameterization used by Cafe for the actual provisioning of the application. The associated Cafe artifacts and their relevance for the MOCCA method are described.

5.1. *Application components*

All components of the sample applications depicted in the architecture diagram (see Fig. 3) have been prototyped (with basic functionality only) by using open source software. To facilitate their provisioning using Cafe a set of properties and `Variability Points` as well as their dependencies were identified. The Input Entry component offers customer interaction through Java Server Pages. It also contains Web service clients implemented in Java to interact with the Good Standing Verifier and the Risk Assessor components. Those clients are initiated from the Java Server Pages and their output is displayed to the customer. The Progress Monitor constitutes another component used for customer interaction. It is also realized as a Java Server Page using a Web service client to obtain a list of all processed orders from the Data Handler component. The good standing of a customer is evaluated by the Good Standing Verifier component which is implemented as a Web service and is called by the Input Entry component. The Risk Assessor component is implemented as a BPEL process, assessing the risk computed by the Good Standing Verifier. If the risk is acceptable, the Risk Assessor initiates the processing of the order by the Order Processing component. The Order Processing component (another BPEL process) handles the actual ordering of an item. First, it verifies that the item is available by accessing the Stock Management component. If so, it removes the item from the stock and stores the order information using the Data Handler component. The Stock Management component is realized as a Web service. It manages the items available to the ordering application. The Data Handler component manages the persistent information about all orders processed. It is also implemented as a Web service.

The general package format for all application components are WAR files with the exception of the BPEL processes which are packaged as ZIP files. The Input Entry, Progress Monitor, and Good Standing Verifier are deployed on Apache Tomcat, the Risk Assessor and Order Processing on Apache ODE, and the Stock Management and Data Handler on JBoss which allows them to access a Hyper SQL Database (HSQLDB) through Hibernate. The required middleware is provided as three virtual machine images. Corresponding to MOCCA's deployment model, the first virtual image contains Apache Tomcat, the second contains Apache ODE, and the third contains JBoss and HSQLDB.

5.2. *Deployment parameterization used by Cafe*

In order to be provisioned using the Cafe environment `Variability Points` of the components have to be identified and made accessible to Cafe. Cafe may access and manipulate any XML file within the component package during the provisioning process. The Cafe metamodel does currently not include `Visible Properties` but it treats them as `Variability Points` that are filled by the provisioning infrastructure. All properties (i.e. instances of `Property`) of the application components are thus transformed into `Variability Points` for the usage in Cafe. Instead of allowing to import `Visible Properties` for a component of a specific type, Cafe allows the import of the `Variability Points` for a component type, i.e. a concrete application server. For the sample application these `Variability Points` are the addresses (e.g. URLs) of the components themselves as well as the addresses of accessed components since they are unknown until after provisioning. For the Item Manager and Data Handler components an additional property is the address of a Hyper SQL Database. To ensure that Cafe can process `Variability Points` they have to be accessible, i.e. may not reside in compiled source files. Web services offered and accessed by the components are therefore configured through WSDL files. During the provisioning process Cafe may adjust the SOAP addresses of the WSDL ports in those files. To configure the database access Hibernate also offers configuration through an XML file that can be set automatically by the Cafe provisioning infrastructure.

5.3. *Provisioning the application using Cafe*

Utilizing MOCCA tools, the application architecture model is modeled and the cloud distribution is computed as described in Sec. 3.3 and based on the labels discussed there. This information is then used to create the Cafe artifacts necessary for provisioning. Semantics and usage of these artifacts have been discussed in Sec. 4.3. Please consider that in practical settings these tools will import existing models.

Since MOCCA and Cafe use the same metamodel regarding the models for deployment topology, automatic installation, and deployment parameterization depicted in Fig. 2, Cafe tools integrate with those of the MOCCA architecture almost seamlessly. Its application model can be obtained from the MOCCA deployment model. However, to provision the sample application this model needs to conform to the provision cluster. Currently, Cafe does not support to overlay an application model with the hyper edges of a cloud distribution to obtain provision clusters. This is due to the fact that Cafe originally does not support the notion of multiple clouds and considers every modeled component to be instantiated only once. Therefore the splitting of middleware components across different clouds, as described by Definitions 3 and 4, has to be made explicit in Cafe's application model. Respective to the provision cluster of the sample application shown in Fig. 13 the Apache Tomcat component is split into two components. One contains the Input Entry component and is hosted in Cloud 1. The other contains the Good Standing Verifier and Progress Monitor components and is hosted in Cloud 2. Other middleware components do not have to be split up since only one single instance is required of those. Note, that a Cafe application model can be derived from a MOCCA deployment model by making the distribution explicit. However, the round-trip back is not possible. Thus the Cafe application model must be extended with information that allows split components to be reassembled again as required by the MOCCA deployment model.

The application model is further used to map components to their implementation by referencing deployment packages such as WAR and ZIP files or OVF Images. Access to `Variability Points` is possible through the definition of XPath

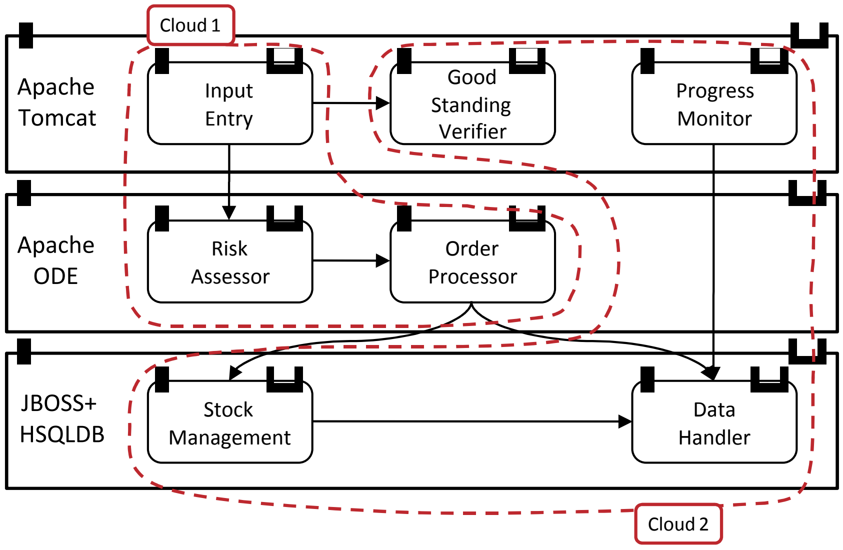


Fig. 13. Provision cluster of the sample application.

expressions altering the XML files inside those packages. A screenshot of the application diagram modeler is shown in the appendix.

In addition to the application model, a so-called component binding is needed. The component binding is used to associate component flows with the components. These are needed to perform the actual provisioning of components as described in Sec. 4.3. They are specified by End Point References through which the flow may be accessed. Since the application model represents the provision cluster, each component may have an individual component flow depending on the cloud to which is it provisioned. A screenshot of the component binding modeler is shown in the appendix.

The final task before provisioning the application is providing the necessary deployment parameterization by specifying the `Variability Points` and their dependencies. The `Variability Points` of the sample application consist of addresses of the Web services offered and accessed by the components. Respecting the dependencies of these `Variability Points`, the provisioning flow for the application is computed automatically. The provisioning flow then performs the provisioning by executing the component flows of the individual components in the right order, respecting the dependencies of components' `Variability Points`. Regarding the variability model created using the variability modeler tool, depicted in appendix the following flow is generated (sketched only in what follows). First middleware components are provisioned which allow the computation of application components' addresses. Circular dependencies are supported by the configuration of components during runtime. Second, the Stock Management, Data Handler, and Good Standing Verifier are provisioned in parallel since they do not depend on any other component. Then the `Variability Points` of the Progress Monitor and Order Processor are known and they can be deployed subsequently. Afterwards, the Risk Assessor and last the Input Entry component are provisioned.

5.4. *Application management in Cafe*

Cafe allows provisioning instances of different applications. To make an application (called application template in Cafe) known to the Cafe system, it must be uploaded to the Cafe system via the Cafe portal. Before uploading, all generated artifacts (code, application model, variability model and component bindings) must be packaged into a Cafe Application Archive (CAR) file. This file can then be uploaded to a Cafe environment. To provision a new instance of an application, a customer selects one of the available templates in the Cafe portal. Then the customer must bind all `Variability Points` that require decisions by the customer. During the binding of the `Variability Points` the customer is guided by the customization flows generated from the `Variability Point`. Once all customer-related `Variability Points` have been bound, the corresponding provisioning flow is executed that makes use of the component flows and the customization flows that bind provisioning-related `Variability Points`, to setup the whole application.

6. Related Work

Given the focus of this paper, related work can be clustered into two main categories: infrastructure topologies and composite application approaches as well as resource optimization approaches in Grids and Clouds.

6.1. *Infrastructure topologies and composite applications*

Several approaches that aim at describing and provisioning complex infrastructure topologie*s* exist. Virtual images as basic building blocks for deploying complex middleware topologies on top of IaaS clouds (Infrastructure as a Service) are investigated in Ref. 19: the conclusion is that individual virtual images are not powerful enough to capture complex infrastructure requirements such as multiple application servers that communicate with authentication servers and databases, for example. Thus, Ref. 19 introduces a so-called "virtual appliance model" that allows defining solutions that are composed of multiple configurable virtual images that can be automatically deployed and run in separate IaaS clouds such as Amazon EC2.[20] The approach presented in Ref. 19 differs from our approach presented in this paper: while Ref. 19 focuses solely on reusable virtual images and their composition, our approach focuses on existing applications and shows how they can be split (in an optimal manner) and moved to (multiple) clouds. Furthermore, virtual images in our approach are optional, i.e. we can also move applications to the cloud without assuming virtual images but deploy applications on top of middleware already available in the cloud. The approach presented in Ref. 19 is based on the approach presented in Ref. 21 which offers a semantically rich metamodel to model complex deployment models similar to those, that we allow to model in MOCCA. However, the difference is, that their approach employs a very rich metamodel while we employ a simple, generic metamodel. While their focus is to capture existing middleware in templates, which requires the complex definition of the template before applications can be modeled. Our model allows to only define those variabilities that are absolutely needed to configure and deploy an application. However, in case complex component types are needed and offered by providers, application modelers can import the variability models of the component types and can integrate them in the application variability model, thus reusing the work of the provider and having the same effect as the approach in Ref. 21.

Elastra[22] and Rightscale[23] offer cloud management services on top of IaaS clouds. Elastra for example provides special languages (ECML, EDML) to describe the infrastructure components of an application such as application servers, databases and their connections. However, concrete resources must be manually assigned to these components, and application modules must be manually installed on top of the components. 3tera[30] offers modeling and deployment of applications based on pre-defined virtual images that can then be deployed in a data center. All of these approaches focus on the management of resources and applications assuming that both are modeled for and managed within a particular target environment. In contrast, our approach is focused on rearranging existing composite applications such that they can be moved to the cloud and can be automatically provisioned across different providers; especially, our approach is independent from particular target environments.

The application packaging standard (APS)[24] focuses on the description of Web applications including, for example, UI components, databases and configuration of Web servers. APS is limited to Web applications and does not allow the annotation of parameters to individual components that would allow the partitioning of the application across multiple Web servers and thus is unsuitable for the purpose of MOCCA. Several approaches exist such as those described in Refs. 25–27 that deal with the description of composite service-oriented applications in terms of composite services that consist of a set of other services. However, all these approaches do not allow the modeling of infrastructure components and thus are unsuitable to describe the infrastructure components of different clouds on which different parts of an application must be deployed.

6.2. *Resource optimization approaches in grids and clouds*

In MOCCA we treat the distribution of application and infrastructure components across different clouds as an optimization problem that solves the task of finding an optimal distribution of components given a set of parameters.

In the Grid[28] the optimization of the distribution and scheduling of jobs across the available resources is an important problem. The component to deal with this optimization is the resource broker or resource scheduler.[28–30] Grid users can send, for instance, job submissions to the broker. The broker then decides which resources will perform the job based on various criteria such as resource utilization, cost of non-functional properties of different resources. Different algorithms exist that optimize the scheduling of the required resources. In Ref. 31, an overview and comparison of these optimization algorithms is given. The problem of job scheduling in a Grid is different from the MOCCA approach, as in MOCCA the distribution across different infrastructure resources is done pre-deployment and not when a request for computation is submitted to an application. Thus the algorithms commonly used in resource brokers focus on the optimization of the distribution of all jobs submitted to a Grid over the available infrastructure. The framework presented in Ref. 32 can act as such a resource broker for application resources in a cloud as it allows to plug-in several algorithms to optimize the distribution of tenants in a cloud application. An approach based on game theory is taken by Lee *et al.*[34] to optimize the allocation of resources for distributed applications in a single cloud given a constrained set of available resources. In Ref. 35, the OpenNebula virtual infrastructure management framework is introduced that includes the Haizea resource scheduling manager to schedule workload across different virtual machines. The Eucalyptus cloud management framework[36] contains similar resource allocation algorithms that allow users of the framework to start and stop virtual machines without having to deal with the concrete resource allocation and scheduling. All these approaches operate at a lower level than the MOCCA approach as they focus on single clouds and Grids and how to optimize the resource allocation in a single cloud or Grid, whereas MOCCA helps architects with the decision how to split applications across different clouds and thus has a different focus. The resource allocation algorithms researched for Grids and clouds can then be used by the individual cloud providers to optimize the resource allocation for the components they have been given to host by the application architects as a result of applying the MOCCA method.

7. Conclusion

In this paper, we proposed a method (the MOCCA method) for solving the move-to-cloud problem. This method has been described in terms of the various steps to be performed and artifacts to be created in order to move an application to the cloud. A metamodel has been presented that formally describes these artifacts and their relations. The artifacts are mainly application models that are enriched by deployment information and labels. The move-to-cloud problem especially subsumes the problem of rearranging the components of an application into groups that might be provisioned into different clouds: this problem has been formalized as an optimization problem (cloud distribution problem) and a solution for solving this problem has been worked out.

We presented the architecture of a tool suite that supports the modeling of the artifacts of the MOCCA method. Also, it supports the automatic derivation of cloud distributions and provision clusters, i.e. it solves the cloud distribution problem. Finally, the provision clusters are automatically provisioned into their target clouds. Thus, the MOCCA method and accompanying tool suite solves the move-to-cloud problem in practical situations. A sample application has been implemented and moved to the cloud to verify the viability of the MOCCA method and tools.

Appendix A. Screenshots

The tool architecture shown in Fig. 12 has been realized based on both, graphical tools as well as tree-based tools (or tree-based tools, respectively). The graphical tools lean more towards domain experts while the tree-based tools are more geared towards supporting expert development personnel. The following subsections show some screenshots of both kinds of tools.

A.1. *Graphical tools*

The graphical tools have been realized based on the Cafe framework,[7] which in turn is based on EMF and GMF. These Eclipse-based technologies support the development of editors for arbitrary metamodels and graphical representations of these. The Cafe metamodel is defined in EMF while GMF is used to specify the graphical representation of metamodel elements.

Figure 14 shows the implemented editor for specifying application models. On the upper left an Eclipse project containing the models and other artifacts, such

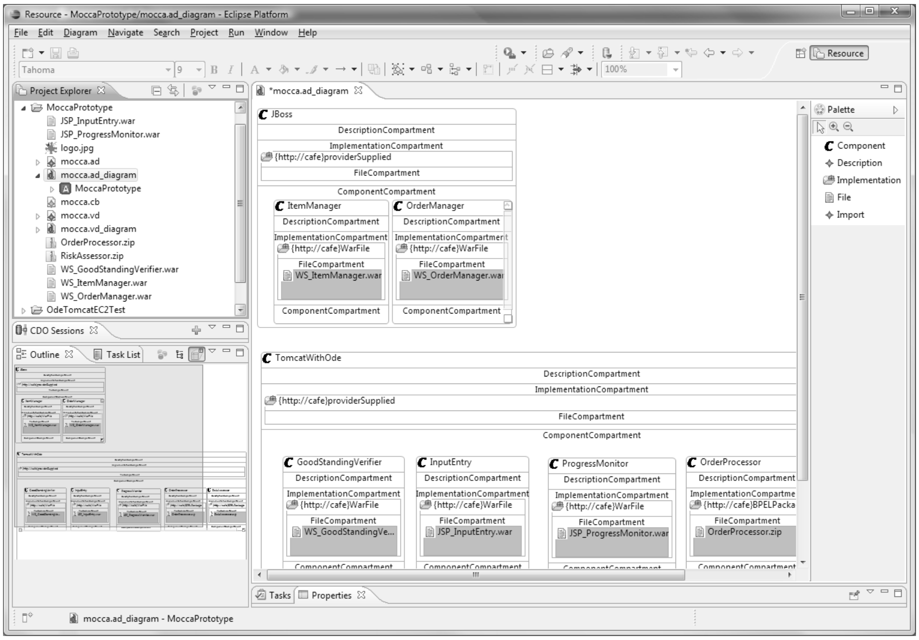


Fig. 14. Modeling the Cafe application diagram.

as code, of the sample application is shown. The representation of the application model is opened in the main view. On the top the JBoss middleware component is modeled. It contains the Stock Manager and Order Manager components. The implementation of the JBoss component is specified as ProviderSupplied. The contained components reference the WAR files which hold their implementation artifacts. These files are also visible in the project explorer on the left side.

`Variability Points` of the components and their dependencies are modeled in the variability diagram shown in Fig. 15. The `Variability Point` of the Apache Tomcat component (TomcatEPR) representing the end point reference of the Apache Tomcat servlet container is visible on the top left in the main view. It has one free `Alternative` as described in Sec. 2.2 which is the hostname of the Apache Tomcat instance. It is filled during the provisioning process. Since the Good Standing Verifier is hosted on the Apache Tomcat component as seen in Fig. 14 the URL of the offered service (represented by the `Variability Point` GSERP) depends on the TomcatEPR `Variability Point`. It is filled once the TomcatEPR `Variability Point` is known. The `Variability Point` of the Input Entry (InputEntryGSEPR) depends on the URL of the Good Standing Verifier

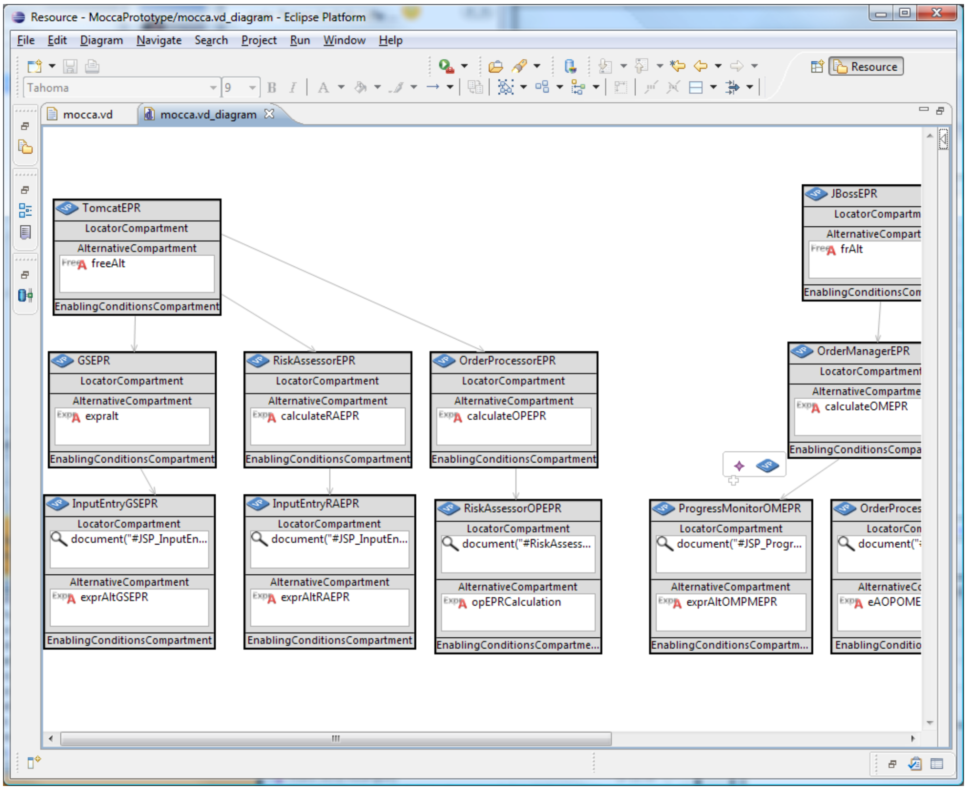
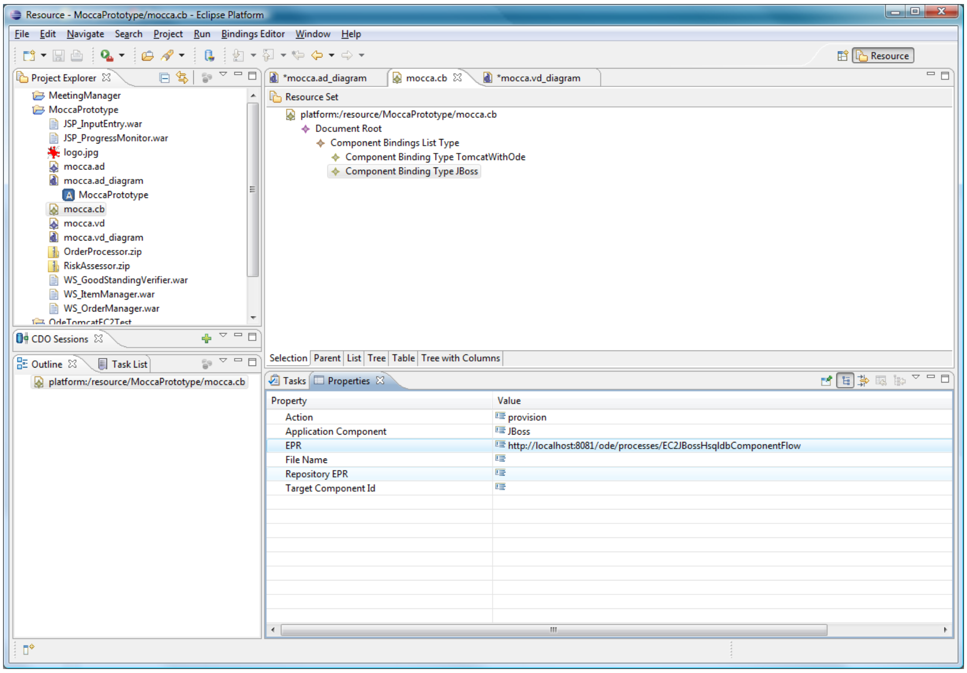


Fig. 15. Modeling the variability diagram of the sample application.


Resource - MoccaPrototype/mocca.cb - Eclipse Platform
File Edit Navigate Search Project Run Bindings Editor Window Help
Resource
Project Explorer
*mocca.ad_diagram
mocca.cb
*mocca.vd_diagram
CDO Sessions
Outline
Task List
Selection Parent List Tree Table Tree with Columns
Tasks
Properties

Project Explorer:

- MeetingManager
- MoccaPrototype
  - JSP_InputEntry.war
  - JSP_ProgressMonitor.war
  - logo.jpg
  - mocca.ad
  - mocca.ad_diagram
    - MoccaPrototype
  - mocca.cb
  - mocca.vd
  - mocca.vd_diagram
  - OrderProcessor.zip
  - RiskAssessor.zip
  - WS_GoodStandingVerifier.war
  - WS_ItemManager.war
  - WS_OrderManager.war
- OdeTomcatEC2Test

Outline: platform:/resource/MoccaPrototype/mocca.cb

Resource Set

- platform:/resource/MoccaPrototype/mocca.cb
  - Document Root
    - Component Bindings List Type
      - Component Binding Type TomcatWithOde
      - Component Binding Type JBoss

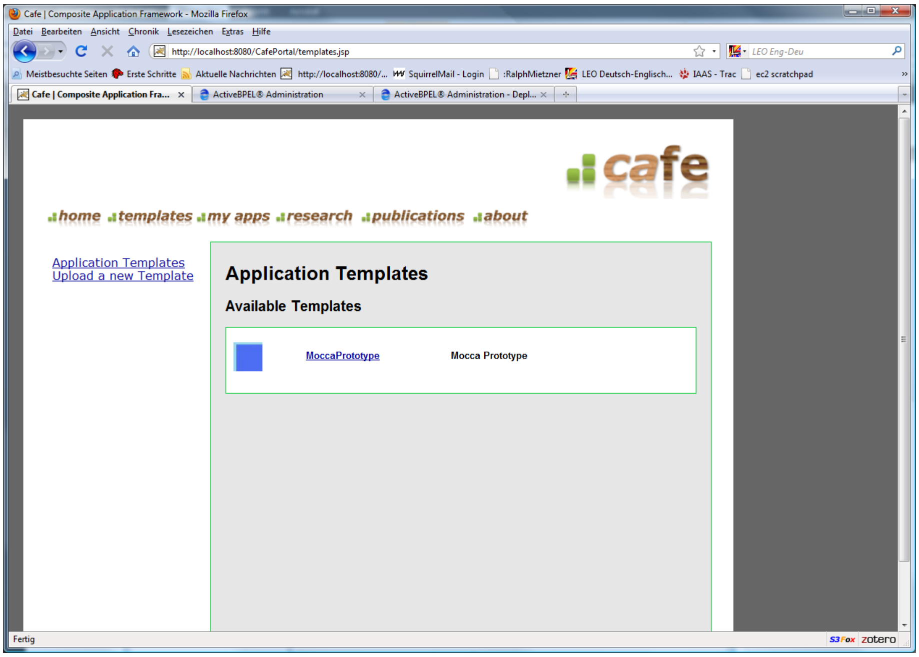
Property	Value
Action	provision
Application Component	JBoss
EPR	http://localhost:8081/ode/processes/EC2JBossHsqldbComponentFlow
File Name	
Repository EPR	
Target Component Id	

Fig. 16. Specifying component bindings.

since it uses this component. From the variability points depicted in this diagram the customization flows can be generated as described in Sec. 4.3. Additionally these dependencies serve as input to the provisioning flow's decision in which order to provision the individual components.

The provisioning flow initiates the component flows of application components to provision an application instance. These component flows are associated with an application component through component bindings. In Fig. 16, the component binding for the JBoss component is selected in the main view of Eclipse. In the properties view on the lower side the properties of this component are displayed. This is where the component JBoss modeled in the application diagram is referenced. The element currently selected in the properties view shows the End Point Reference of the component flow for that referenced component. The action element specifies the purpose for which the component flow is used. In this case it is for provisioning. Additional component bindings may also be specified to reference component flows implementing other actions. Those include adding other components after provisioning the referenced component or deprovisioning it.

Having packaged the sample application using the CAR export file wizard of the application modeler, the application can be uploaded to the Cafe Portal. Having uploaded the application it becomes available under the templates tab (as shown in Fig. 17) and customers can subscribe to that application.


Cafe | Composite Application Framework - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://localhost:8080/CafePortal/templates.jsp
LEO Eng-Deu
Meistbesuchte Seiten Erste Schritte Aktuelle Nachrichten http://localhost:8080/... SquirrelMail - Login :RalphMietzner LEO Deutsch-Englisch... IAAS - Trac ec2 scratchpad
Cafe | Composite Application Fra...
ActiveBPEL® Administration
ActiveBPEL® Administration - Depl...
cafe
home templates my apps research publications about
Fertig
zotero

Application Templates
Upload a new Template

Application Templates

Available Templates

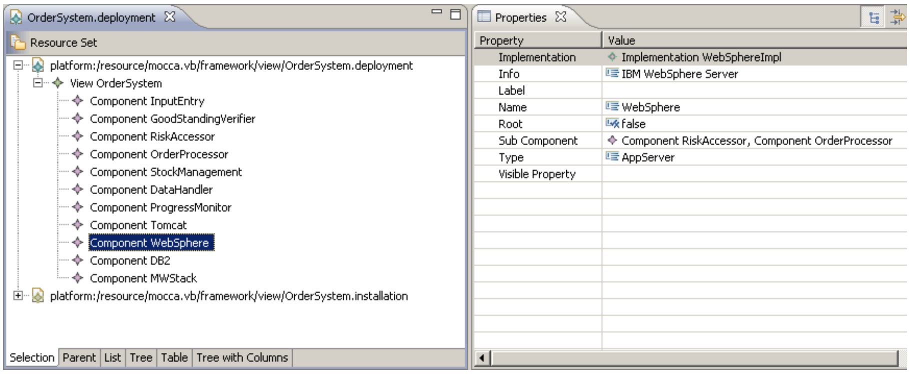
MoccaPrototype Mocca Prototype

Fig. 17. The sample application in the Cafe portal.

A.2. *Tree-based tools*

The tree-based tools have been realized by using VBMF,[11] which is an Eclipse-based tool supporting model driven development of applications. Especially, it supports an easy way to specify metamodels and generates corresponding tree-based modeling tools. Based on Fig. 2, the metamodels denoted as "model types" in that figure have been specified in VBMF and the corresponding modeling tools for tree-based specification of architecture models, deployment topology etc. result.

Figure 18 shows the tool allowing to model components of an architecture model (the "boxes") and also of deployment models. The *WebSphere* component is



platform:/resource/mocca.vb/framework/view/OrderSystem.deployment
View OrderSystem
Component InputEntry
Component GoodStandingVerifier
Component RiskAccessor
Component OrderProcessor
Component StockManagement
Component DataHandler
Component ProgressMonitor
Component Tomcat
Component WebSphere
Component DB2
Component MWStack
platform:/resource/mocca.vb/framework/view/OrderSystem.installation

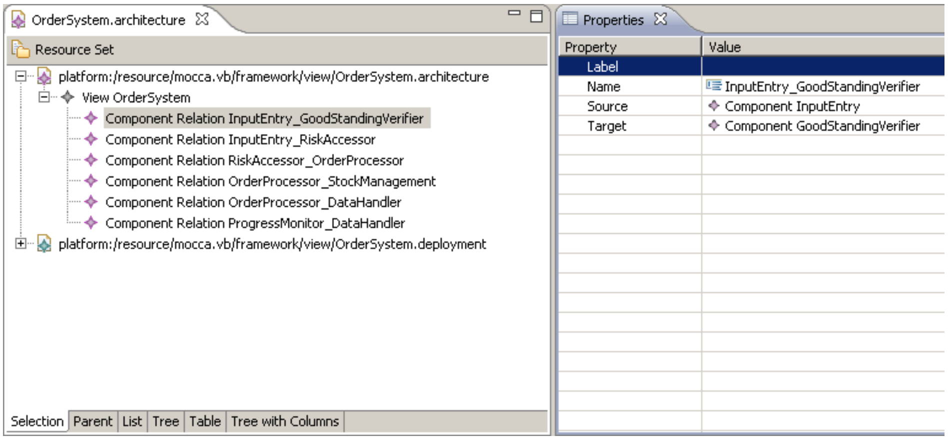
Property	Value
Implementation	Implementation WebSphereImpl
Info	IBM WebSphere Server
Label	
Name	WebSphere
Root	false
Sub Component	Component RiskAccessor, Component OrderProcessor
Type	AppServer
Visible Property	

Fig. 18. Modeling components.

high-lighted in the left part of the screen and the right part shows the properties of that component, e.g. its name and its type (in this case "AppServer"). The sub-components of *WebSphere* application server are the *Risk Assessor* as well as the *Order Processor* components (see Fig. 4).

The "arrows" of an architecture model are specified as component relations as shown in Fig. 19. The left side of the screenshot high-lights the component relation *InputEntry_GoodStandingVerifier* and the right side show its properties especially that it begins at the Input Entry component and ends at the Good Standing Verifier component (as shown in Fig. 3).

The fact, that the *MWStack* component encompasses (amongst others) the *WebSphere* component and the *DB2* component (see Fig. 5) has been specified in Fig. 20.



OrderSystem.architecture

Resource Set

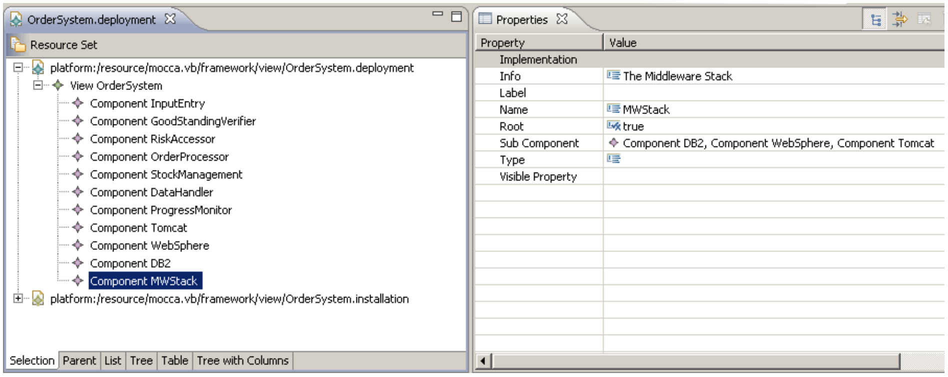
```
platform:/resource/mocca.vb/framework/view/OrderSystem.architecture
  View OrderSystem
    Component Relation InputEntry_GoodStandingVerifier
    Component Relation InputEntry_RiskAccessor
    Component Relation RiskAccessor_OrderProcessor
    Component Relation OrderProcessor_StockManagement
    Component Relation OrderProcessor_DataHandler
    Component Relation ProgressMonitor_DataHandler
platform:/resource/mocca.vb/framework/view/OrderSystem.deployment
```

Selection | Parent | List | Tree | Table | Tree with Columns

Properties

Property	Value
Label	
Name	InputEntry_GoodStandingVerifier
Source	Component InputEntry
Target	Component GoodStandingVerifier

Fig. 19. Modeling relations.



OrderSystem.deployment

Resource Set

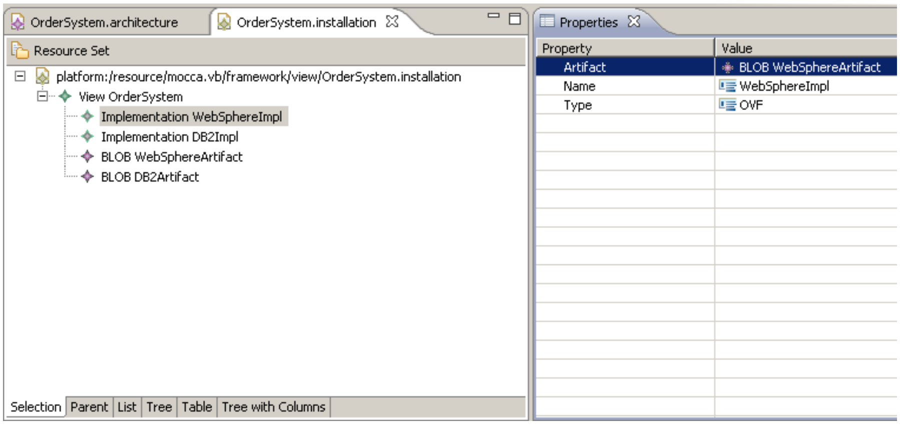
```
platform:/resource/mocca.vb/framework/view/OrderSystem.deployment
  View OrderSystem
    Component InputEntry
    Component GoodStandingVerifier
    Component RiskAccessor
    Component OrderProcessor
    Component StockManagement
    Component DataHandler
    Component ProgressMonitor
    Component Tomcat
    Component WebSphere
    Component DB2
    Component MWStack
platform:/resource/mocca.vb/framework/view/OrderSystem.installation
```

Selection | Parent | List | Tree | Table | Tree with Columns

Properties

Property	Value
Implementation	
Info	The Middleware Stack
Label	
Name	MWStack
Root	true
Sub Component	Component DB2, Component WebSphere, Component Tomcat
Type	
Visible Property	

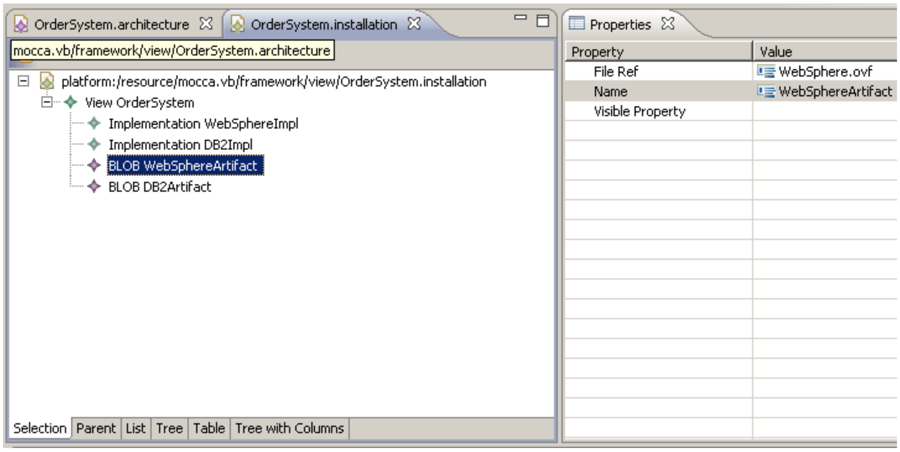
Fig. 20. Modeling composite components.



platform:/resource/mocca.vb/framework/view/OrderSystem.installation
- View OrderSystem
  - Implementation WebSphereImpl
  - Implementation DB2Impl
  - BLOB WebSphereArtifact
  - BLOB DB2Artifact

Property	Value
Artifact	BLOB WebSphereArtifact
Name	WebSphereImpl
Type	OVF

Fig. 21. Specifying implementations.

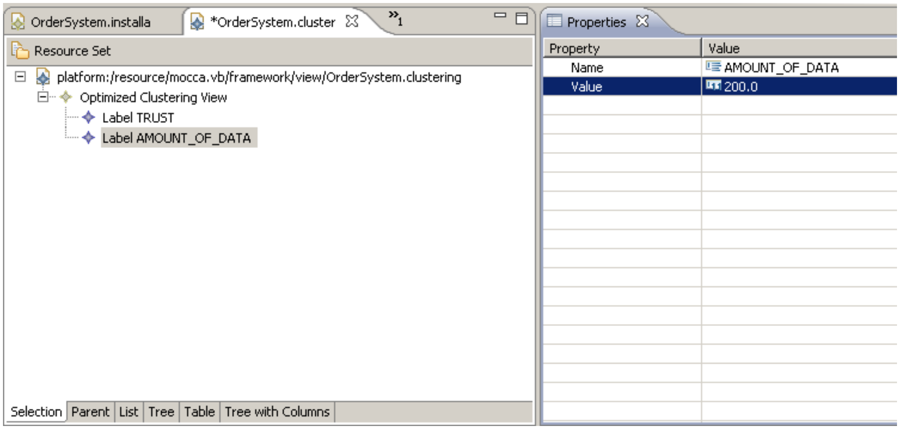


mocca.vb/framework/view/OrderSystem.architecture

platform:/resource/mocca.vb/framework/view/OrderSystem.installation
- View OrderSystem
  - Implementation WebSphereImpl
  - Implementation DB2Impl
  - BLOB WebSphereArtifact
  - BLOB DB2Artifact

Property	Value
File Ref	WebSphere.ovf
Name	WebSphereArtifact
Visible Property	

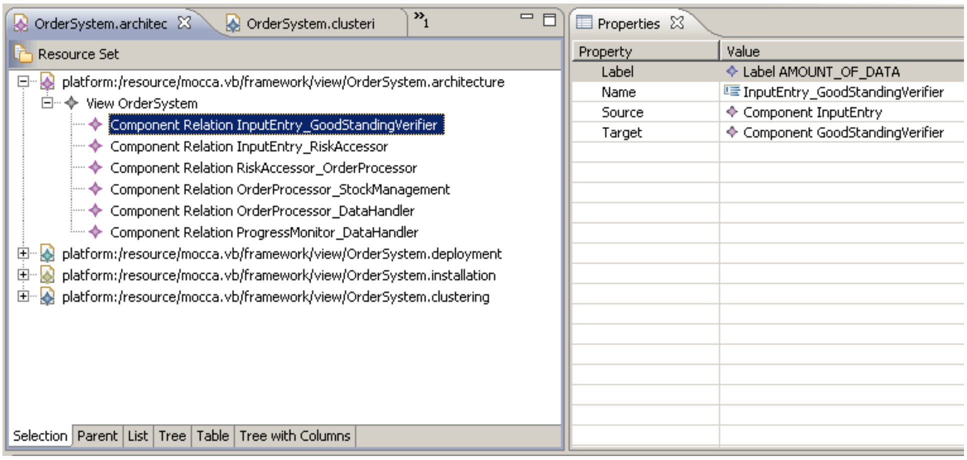
Fig. 22. Specifying artifacts.



platform:/resource/mocca.vb/framework/view/OrderSystem.clustering
- Optimized Clustering View
  - Label TRUST
  - Label AMOUNT_OF_DATA

Property	Value
Name	AMOUNT_OF_DATA
Value	200.0

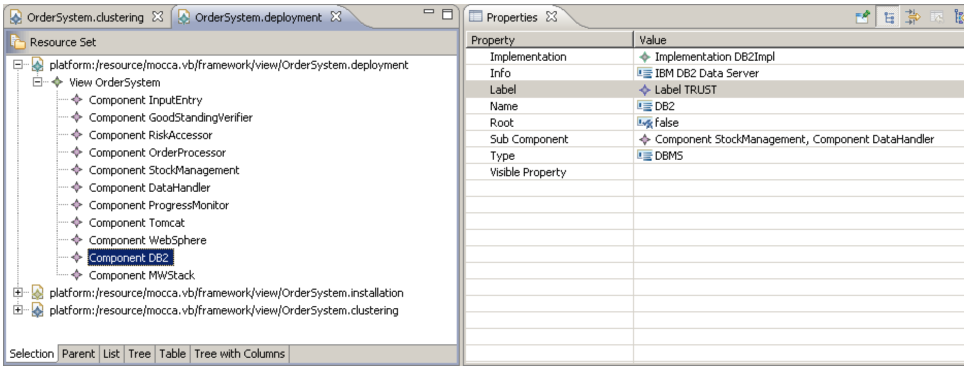
Fig. 23. Modeling labels.



Resource Set
- platform:/resource/mocca.vb/framework/view/OrderSystem.architecture
  - View OrderSystem
    - Component Relation InputEntry_GoodStandingVerifier
    - Component Relation InputEntry_RiskAccessor
    - Component Relation RiskAccessor_OrderProcessor
    - Component Relation OrderProcessor_StockManagement
    - Component Relation OrderProcessor_DataHandler
    - Component Relation ProgressMonitor_DataHandler
- platform:/resource/mocca.vb/framework/view/OrderSystem.deployment
- platform:/resource/mocca.vb/framework/view/OrderSystem.installation
- platform:/resource/mocca.vb/framework/view/OrderSystem.clustering

Property	Value
Label	Label AMOUNT_OF_DATA
Name	InputEntry_GoodStandingVerifier
Source	Component InputEntry
Target	Component GoodStandingVerifier

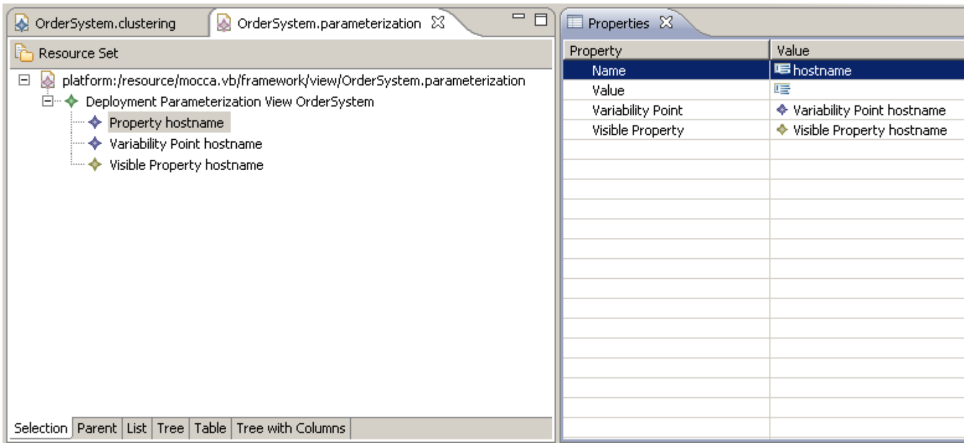
Fig. 24. Labeling relations.



Resource Set
- platform:/resource/mocca.vb/framework/view/OrderSystem.deployment
  - View OrderSystem
    - Component InputEntry
    - Component GoodStandingVerifier
    - Component RiskAccessor
    - Component OrderProcessor
    - Component StockManagement
    - Component DataHandler
    - Component ProgressMonitor
    - Component Tomcat
    - Component WebSphere
    - Component DB2
    - Component MWStack
- platform:/resource/mocca.vb/framework/view/OrderSystem.installation
- platform:/resource/mocca.vb/framework/view/OrderSystem.clustering

Property	Value
Implementation	Implementation DB2Impl
Info	IBM DB2 Data Server
Label	Label TRUST
Name	DB2
Root	false
Sub Component	Component StockManagement, Component DataHandler
Type	DBMS
Visible Property	

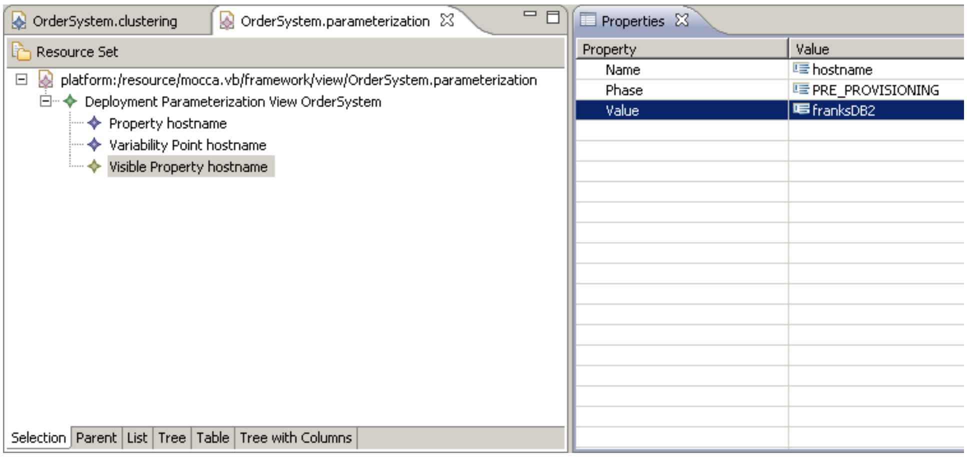
Fig. 25. Labeling components.



Resource Set
- platform:/resource/mocca.vb/framework/view/OrderSystem.parameterization
  - Deployment Parameterization View OrderSystem
    - Property hostname
    - Variability Point hostname
    - Visible Property hostname

Property	Value
Name	hostname
Value	
Variability Point	Variability Point hostname
Visible Property	Visible Property hostname

Fig. 26. Modeling properties.



platform:/resource/mocca.vb/framework/view/OrderSystem.parameterization
Deployment Parameterization View OrderSystem
Property hostname
Variability Point hostname
Visible Property hostname

Property	Value
Name	hostname
Phase	PRE_PROVISIONING
Value	franksDB2

Fig. 27. Modeling visible properties.

The information from Fig. 5 that *WebSphere* is realized by an implementation of type *OVF* and that this OVF file is a `BLOB Artifact` having a certain `FileRef` attribute is given in Figs. 21 and 22, respectively.

Figure 23 gives an example of modeling labels. The label *AMOUNT_OF_DATA* high-lighted and got the value 200 assigned. This label has been associated to the component relation *InputEntry_GoodStandingVerifier* in Fig. 24. The other label called *TRUST* has been associated with the *DB*2 component in Fig. 25.

The specification of properties is depicted in Fig. 26: the property *hostname* has been shown that has no value assigned (according to the situation modeled in Fig. 5). Figure 27 also specifies *hostname* as a visible property now having the value *franksDB*2 (see Fig. 5).

References

1. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, Above the Clouds: A Berkeley View of Cloud Computing, Technical Report (2009).
2. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, *Future Generation Computer Systems* **25** (2009) 599–616.
3. S. Jha, A. Merzky and G. Fox, Using clouds to provide grids higher-levels of abstraction and explicit support for usage modes, Concurrency and computation: Practice and experience, Special Issue of the OGF (2009).
4. National Institute of Standards and Technology, Draft NIST Working Definition of Cloud Computing (2009), http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v12.doc.
5. Open Cloud Manifesto (2009), http://www.opencloudmanifesto.org/Open%20Cloud%20Manifesto.pdf.
6. F. Leymann, Cloud computing: The next revolution in IT, in *Proc. of the 52th Photogrammetric Week* (Stuttgart, Germany, 2009).

7. R. Mietzner, T. Unger and F. Leymann, Cafe: A generic configurable customizable composite cloud application framework, in *Proc. of the 17th Intl. Conf. on Cooperative Information Systems*, CoopIS (Springer-Verlag, Berlin, Heidelberg, 2009).
8. S. Weerawarana, F. Curbera, F. Leymann, T. Storey and D. F. Ferguson, *Web Services Platform Architecture* (Prentice Hall, 2005).
9. DMFT Standard: Open Virtualization Format Specification, Document Number: DSP0243 (2009), http://www.dmtf.org/standards/published_documents/DSP0243 _ 1.0.0.pdf.
10. The ACME Project, http://www.cs.cmu.edu/∼acme/.
11. H. Tran, U. Zdun and S. Dustdar, View-based and model-driven approach for reducing the development complexity in process-driven SOA, in *Proc. of the Intl. Conf. on Business Processes and Services Computing* (Leipzig, Germany, 2007).
12. BPMN 1.1, http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf.
13. L. Hohmann, *Beyond Software Architecture* (Addison-Wesley, 2003).
14. M. R. Garey and D. S. Johnson, *Computers and Intractability — A Guide to the Theory of NP-Completeness* (W.H. Freeman & Co., 1990).
15. S. Kirkpatrick, C. Gelatt and M. Vecchi, Optimization by simlated annealing, *Science* **220** (1983).
16. O. Danylevych, D. Karastoyanova and F. Leymann, Optimal stratification of transactions, in *Proc. of the 4th Intl. Conf. on Internet and Web Applications and Services* (2009), pp. 493–498.
17. R. Mietzner and F. Leymann, Generation of BPEL customization processes for SaaS applications from variability descriptors, in *Proc. of the Intl. Conf. on Services Computing* (Washington, DC, USA, 2008).
18. BPEL, www.oasis-open.org/committees/wsbpel/.
19. A. Konstantinou, T. Eilam, M. Kalantar, A. Totok, W. Arnold and E. Snible, An architecture for virtual solution composition and deployment in infrastructure clouds, in *Proc. of the 3rd Intl. Workshop on Virtualization Technologies in Distributed Computing* (VTDC, 2009).
20. Amazon EC2, http://aws.amazon.com/ec2/.
21. K. El Maghraoui, A. Meghranjani, T. Eilam, M. H. Kalantar and A. V. Konstantinou, Model driven provisioning, bridging the gap between declarative object models and procedural provisioning tools, in *Proc. of the 7th Intl. Middleware Conference* (2006), pp. 404–423.
22. ELASTRA, http://www.elastra.com/.
23. RightScale, http://www.rightscale.com/.
24. SWSoft Inc. Application Packaging Standard (APS) (2007), http://apsstandard.com/r/doc/package-format-specification-1.0.pdf.
25. B. Benatallah, M. Dumas and Q. Z. Sheng, Facilitating the rapid development and scalable orchestration of composite web services, *Distributed and Parallel Databases* **17**(1) (2005) 5–37.
26. Open SOA Collaboration (OSOA), SCA Service Component Architecture, Assembly Model Specification Version 1.00 (2007), http://www.osoa.org/download/attachments/35/SCA_ AssemblyModel_ V100.pdf.
27. F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic and S. Dustdar, Towards composition as a service — a quality of service driven approach, in *Proc. of the Intl. Conf. on Data Engineering* (Shanghai, China, 2009), pp. 1733–1740.
28. I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure* (Morgan Kaufmann, 2004).
29. R. Buyya, D. Abramson and J. Giddy, Nimrod-G resource broker for service-oriented grid computing, *IEEE Distributed Systems Online* **2**(7) (2001).

30. 3tera, http://www.3tera.com/.
31. C. Yang, P. Shih and K. Li, A high-performance computational resource broker for grid computing environments, in *Proc. of the 19th Intl. Conf. on Advanced Information Networking and Applications*, Vol. 2, IEEE Computer Society (Washington, DC, USA), pp. 333–336.
32. K. Li, Job scheduling and processor allocation for grid computing on metacomputers, *Journal of Parallel and Distributed Computing* **65**(11) (2005) 1406–1418.
33. C. Fehling, F. Leymann and R. Mietzner, A framework for optimized distribution of tenants in cloud applications, in *Proc. of the 3rd International Conference on Cloud Computing* (Miami, USA, 2010), pp. 252–259.
34. C. Lee, J. Suzuki, A. Vasilakos, Y. Yamamoto and K. Oba, An evolutionary game theoretic approach to adaptive and stable application deployment in clouds, in *Proc. of the 2nd Workshop on Bio-Inspired Algorithms for Distributed Systems* (ACM, New York, NY, 2010), pp. 29–38.
35. B. Sotomayor, R. S. Montero, I. M. Llorente and I. Foster, Virtual infrastructure management in private and hybrid clouds, *IEEE Internet Computing* **13**(5) (2009) 14–22.
36. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, The eucalyptus opensource cloud-computing system, in *Proc. of the 9th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid-Volume* (IEEE Computer Society, 2009), pp. 124–131.