Component-Oriented Programming and Datacenter Applications

Venkatesh-Prasad Ranganath
Microsoft Research India
rvprasad@microsoft.com

Abstract

Component-oriented programming enables the *development* of easily deployable and manageable applications while datacenters supports easy *deployment* and *management* of applications (via provisioning). So, why not consider component-oriented programming as the paradigm to program datacenter applications?

1. Introduction

Recently, datacenters have been gaining focus as an enterprise application platform because of (1) the decreased cost and increased speed of provisioning virtual machines and networks and (2) the catering of basic services (ranging from large scale storage to e-mail) as integral features of the runtime environment. Consequently, there will be a need for approaches to develop *datacenter applications (DAs)*; specifically, approaches that enable aggressive use of provisioning capabilities of datacenters to easily deploy and manage DAs.

For over a decade, component-based applications have been prevalent in the form of applications based on COM, .NET, EJB, and Web services. A *component-based application (CBA)* is a composition of services published by various components along with various component specific and application specific provisioning requirements/constraints. The paradigm of programming CBAs through *component development, composition creation, component deployment,* and *composition assembly* is commonly referred to as *Component-oriented programming (COP)* (2). Some of the key features of COP, such as statelessness of components, crisp separation between interface and implementation and between development and deployment, and the declarative specification of components and compositions, enable the use of *Model Driven Development (MDD)* techniques to develop CBAs.

As the features of COP complement the capabilities of datacenters, we opine that component-oriented programming is a good paradigm to program datacenter applications. In this paper, we justify our opinion by comparing the tasks (and artifacts) involved in COP and programming DAs.

2. Component-Oriented Programming (COP)

A *component* is a composable and transparent software element that provides and uses services via well-defined interfaces, supports independent versioning and deployment, and caters to multiple mutually independent simultaneous uses. These characteristics of a component enable high degree of reusability and implementation variability.[1] A *component-based applications (CBA)* is a composition of services published by various components along with various component specific and application specific provisioning requirements/constraints. Typically, any application logic specific code is encapsulated in one or more of the participating components (if needed, new components are created). A *container* defines the execution environment for components and shields them from the actual runtime environment. Typically, it hosts components, provides basic services (such as communication and persistence) to the hosted components, and manages the interaction between the components and the runtime environment.[2]

The above COP entities are constrained as follows: (1) a container can host more than one instance of a component, (2) a container can host instances of different components, (3) a container is confined to a single physical node/host, and (4) an instance of a component resides entirely in a container.

We shall now describe various tasks involved in COP.

2.1 Component Development

The task of developing a component entails *programming a component and packaging it for deployment in a container.* After identifying the collection of closely related services that constitute a component, following are the typical steps involved in developing a component.

- *Define interfaces to expose the provided services.* Most often, interfaces are defined in high level languages (such as IDL) to enable the interoperability of components independent of the underlying component implementation.
- *Program a component implementation to realize the services.* Besides the actual programming, a component implementation requires a mapping of the interface definition to implementation features that realize the interface. Typically, such a mapping is generated along with a skeletal implementation by a programming language and runtime specific interface definition language compiler. Hence, the actual activity in programming a component implementation is the addition of the business logic to the generated skeletal implementation.
- *Associate unique identifiers with the component and the exposed interfaces.* These identifiers enable the transparent publication and discovery of components and provided services (via the corresponding interfaces).
- *Identify any required external service dependences.* To ease component development, most of the basic services such as communication and persistence are externalized and provided by containers. Typically, these services are available as part of the controlled runtime sandbox provided by the containers to the hosted components.
- *Package the component implementation.* Besides the executable implementation, the package will also contain any required

[1] By *implementation variability*, we mean that an implementation X of component C can be replaced by an implementation Y of component C with almost no impact on the application.

[2] A component and its instance can be thought of as a type and a value of that type, respectively. We shall use the term *component* to mean a component type. When unambiguous, we shall abuse the term to mean a component instance.

libraries along with a *deployment descriptor* that contains the metadata required to deploy and publish the component and its interfaces in the runtime environment.

2.2 Composition Creation

As in programming an application by composing features of various libraries, a component-based application is programmed by composing services provided by various components. Hence, the composition creation tasks involves the following composition-oriented activities.

- *Identify components that provide necessary services.* This is usually specified in terms of the published component and interface identifiers. Identifier-based specification enables the runtime system to dynamically provision components that cater the requested services. Consequently, the composition is more manageable as a result of the increased flexibility in provisioning.
- *Connect the interfaces of components to realize the composition.* Such a composition is merely an inter-service data and control flow network that realizes the intent of the application. Typically, any application logic not present in any of the off-the-shelf components are bundled into application specific components and used in the composition.
- *Reason about the correctness of the composition.* Similar to programs in C and other languages, compositions are prone to structural (syntactic) and behavioral (semantic) errors. Simple compile-time techniques such as type checking are employed to weed out basic structural errors while more sophisticated techniques are employed to perform deeper reason about richer or domain specific properties (such as connectivity between services and connectivity guarantees) of the composition. Such deeper reasoning most often depends on properties of the runtime environment and can generate new constraints on the runtime environment.

  Also, such reasoning can depend on the specifications of the internals of the participating components. If such specifications are exposed, then model driven development can be employed to iteratively refine models of components and compositions and, consequently, alleviate the reasoning burdens associated with large composition and/or rich property spaces (1).
- *Identify provisioning constraints.* Non-functional features of the participating components such as availability, response time, and resource requirements cannot be programmed into components. Instead, the requirements to realize these features are identified as provisioning constraints on the runtime environment.
- *Create an assembly descriptor.* Unlike a component package, an assembly descriptor is created as the final artifact of composition creation. Typically, an assembly descriptor declaratively specifies the required components (via identifiers), the connection between various interfaces of components, and any provisioning constraints on the runtime environment either in the context of specific participating components or the composition.

2.3 Component Deployment

The deployment of a component involves the installation of the contents of a component package into a container that provides the requested basic services and publication of the identifiers of the component and its published services with the discovery service of the runtime system.

2.4 Composition Assembly

Unlike the deployment of the component, assembling a composition is more involved. Once the components involved in an assembly are located in the runtime environment, they are connected in accordance with the assembly and the application is available for use.

However, it may be impossible to locate required component-container combinations that satisfy required provisioning constraints. In such situations, a static runtime system can either abort the assembling of the composition while a dynamic runtime system can try to identify or even create appropriate containers, which satisy the provisioning constraints, to deploy the components and then retry to assemble the composition. With MDD, these situations can be avoided by reasoning about the combination of the models of the composition and the runtime environment.

3. Programming Datacenter Applications (DAs)

A *datacenter* can be perceived as a computing system composed of a large number of well connected computers. Given this view, datacenter applications can be perceived as a distributed application that executes on multiple computers.

Beyond the above perception, a datacenter can be easily configured into smaller dedicated virtual computing systems by means of virtual networking and virtualization technologies. Given this capability of a datacenter, a datacenter application can be viewed as a composition of *application fragments (AFs)* hosted on *virtual machines (VMs)* in a dedicated virtual network. Hence, the deployment of a datacenter application can be perceived as the deployment of the virtual machines that host the application fragments, setting up the virtual network, and connecting the application fragments to other application fragments and the services provided by the datacenter (in accordance to the application composition). Depending on the provisioning constraints of the application fragments, the deployment may involve the selection of existing instances of AFs, existing VMs to host new instances of AFs, powered-up computers to host corresponding VMs, or even power-up computers to host corresponding VMs. The provisioning capability of the datacenter enables this approach of deploying and managing applications.

4. Relevance of COP to Program DAs

Based on the above description of programming DAs and earlier description of COP, we can draw parallels between concepts/entities in COP and in programming DAs: component and application fragments, composition and application, and containers and virtual machines. Similar parallels can be drawn between the approach to develop and deploy a datacenter application and the COP approach to create and assemble a composition.

Given the strong correlation of entities and tasks, we opine that it may be worthwhile to explore component-oriented programming (1) as a paradigm to program datacenter applications and (2) to identify and leverage (avoid) approaches and techniques that can facilitate (inhibit) the programming of datacenter applications.

References

[1] John Hatcliff, William Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems." In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003.

[2] Clemens Szyperski. "Component Software: Beyond Object-Oriented Programming." Addison-Wesley, 2002.