國立交通大學

資訊科學與工程研究所

碩 士 論 文

自動服務部署，組合及驗證平台之設計與實作

A Unified Framework for Service Deployment, Composition, and Validation

研 究 生：莊景棠

指導教授：陳俊穎　教授

中 華 民 國 九 十 六 年 四 月

自動服務部署，組合及驗證平台之設計與實作
A Unified Framework for Service Deployment, Composition, and Validation

研 究 生：莊景棠　　Student：Jing-Tang Zhuang

指導教授：陳俊穎　　Advisor：Jing-Ying Chen

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

April 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年四月

國　立　交　通　大　學

博碩士論文全文電子檔著作權授權書

（提供授權人裝訂於紙本論文書名頁之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程系所_____組，95學年度第 2 學期取得碩士學位之論文。

論文題目：自動服務部署，組合及驗證平台之設計與實作

指導教授：陳俊穎

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：

本校及台灣聯合大學系統區域網路	■ 立即公開
校外網際網路	■ 立即公開

授 權 人：莊景棠

親筆簽名： 莊景棠

中華民國 96 **年** 4 **月** 12 **日**

國　立　交　通　大　學

博碩士紙本論文著作權授權書

（提供授權人裝訂於全文電子檔授權書之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程系所_____組，95 學年度第 2 學期取得碩士學位之論文。

論文題目：自動服務部署, 組合及驗證平台之設計與實作

指導教授： 陳俊穎

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：_____，請將論文延至____年____月____日再公開。

授 權 人：莊景棠

親筆簽名： 莊景棠

中華民國 96 **年** 4 **月** 12 **日**

國家圖書館博碩士論文電子檔案上網授權書

ID:GT009223592

本授權書所授權之論文為授權人在國立交通大學資訊科學與工程系所 95 學年度第 2 學期取得碩士學位之論文。

論文題目：自動服務部署,組合及驗證平台之設計與實作

指導教授：陳俊穎

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授權人：莊景棠

親筆簽名：莊景棠

民國 96 **年** 4 **月** 12 **日**

國立交通大學

研究所碩士班

論文口試委員會審定書

本校 資訊科學與工程 研究所 莊景棠 君

所提論文: A Unified Framework for Service Deployment, Composition and Validation

自動服務部署, 組合及驗證平台之設計與實作

合於碩士資格水準、業經本委員會評審認可。

口試委員：陳俊穎 _____

楊武 廖珏洲

指導教授：陳俊穎

所 長：曾文貴

中 華 民 國 九十六 年 一 月 二十六 日

自動服務部署，組合及驗證平台之設計與實作

學生：莊景棠　　指導教授：陳俊穎 博士

國立交通大學資訊科學與工程研究所

摘　　要

服務導向運算是近來一項廣受矚目的重要技術，其目標在建立一個廣泛跨網路的運算資源整合平台，使得個人及公司團體能夠方便取得各種軟體服務，並在線上將它們組裝成服務導向的應用。然而，發展服務導向應用的過程不僅包含軟體服務的取得和組裝，還包括一般軟體工程上重要的步驟，像是分析、設計、測試、部署或是程式執行時的管理。然而，現有服務導向架構的設計大都不將這些步驟納入考量。為了能夠更確實地實現以元件為基礎的服務導向運算目標，我們認為上述許多重要的軟體發展步驟必須進一步中立化及標準化。有鑑於此，我們提出一個新的服務導向的架構，將服務管理的角色標準化，使得軟體服務的發展、組合和執行時的管理都可以一併處理，不但能進一步加速軟體發展的流程，也能避免過度仰賴特定的實作技術或廠商。在此架構基礎上，我們也發展了一些自動服務組合和測試的工具，來驗證我們的想法。

A Unified Framework for Service Deployment, Composition, and Validation

Student : Jing-Tang Zhunag Advisors : Dr. Jing-Ying Chen

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Service-oriented computing is the latest technology innovation aiming at establishing a universal interoperability platform on top of which individuals and companies can acquire and assemble reusable services into service-oriented applications easily. However, developing service-oriented applications involves not only service acquisition and composition, but also other important software engineering activities such as analysis, design, testing, deployment, or even run-time management. Existing service-oriented architectures such as Web Services fall short when these supporting activities are concerned. To truly realize the component principle underpinning service-oriented computing, we argue that standards and conventions are needed to facilitate most, if not all of these activities in platform- and vendor-neutral ways. To tackle this fundamental problem, we propose a generic service-oriented architecture which standardizes the role of service containers, so that service development, composition, and run-time management can also be expressed, making it possible to streamline development process with minimized vendor dependencies. Based on the architecture, we also develop a framework that permits automatic service composition and verification.

誌　　謝

對於學位論文的完成，首先必須感謝我的指導教授陳俊穎老師，在求學的過程中以及研究遭遇瓶頸時，總是耐心的給予我指導，不但指引我正確的方向，對於思考解決問題的方法和態度上，也使我獲益良多；同時特別感謝口試委員楊武教授與廖珧洲教授在百忙之中給予論文許多寶貴的指導與意見，使得論文的內容更加完備，在此特別感謝。

此外，也要感謝研究室的伙伴們，嘉源、君翰、亦秋、嘉宏、以及學弟們，在研究進行時給與我許多的支持與鼓勵，並陪伴我度過研究生涯。還要感謝建宏學長、舜禹學長、訓宏學長、以及許吉學長，在遭遇問題時總是不吝給予我建議與協助，並提供寶貴的研究經驗。同時也感謝我的朋友，在我快堅持不下去的時候，一直給我加油打氣與支持，這都是我能堅持下去的力量來源。

最後，由衷地感謝我最親愛的家人，由於他們的支持與包容，提供一個無後顧之憂的環境，讓我得以順利的完成學業，願將這份榮耀獻給我的家人。

莊景棠 謹誌 2007 年 4 月

於交通大學研究生室

Table of Contents

List of Figures

Chapter 1. Introduction

Service-oriented computing (SOC) is becoming the most prominent distributed computing paradigm recently, attempting to establish a component-based infrastructure on top of which service providers and application developers can develop and deploy self-contained, reusable services, and combine these services to form larger services or service-oriented applications. Today, many approaches to service-oriented architecture (SOA) have been proposed, each realizing SOC differently. However, they all need to support service composition sufficiently in order to make it easy for developer to adapt or compose services into larger services or applications with relatively less efforts. With this property, SOC promises to offer companies the flexibility and agility they need – both are crucial factors in current IT industry where requirements changes are frequent and time to market pressure is high.

Still, developing service-oriented applications involves more than simply obtaining services and snapping them together. To deliver a final system that meets what end users want, other important software engineering activities including requirements engineering, analysis, design, testing, deployment, and even run-time management are still required. On the other hand, most SOC approaches emphasize on enabling mechanisms in a bottom-up fashion. For example, the Web Services [1] protocol stack as represented by XML, SOAP, and WSDL are the foundation of the Web Services architecture, based on which higher-level standards for service composition or orchestration such as WS-BPEL [2] from OASIS or WS-CDL from W3C are developed. Such a layered, bottom-up architecture design is also common in many other distributed computing platforms. However, such design also has consequences that may go against the component principle behind SOC, where people are supposed to be able to flexibly and conveniently assemble services into useful applications for their own use, regardless how these services are built and on which platforms they are run. When multiple approaches to service composition are allowed, for example, both service providers and consumers need to “take sides,” and will gradually fragment the service market into multiple camps with hard-to-cross boundaries. This issue cannot be resolved by simply asking service providers to provide the same kinds of services for different composition approaches. First of all, doing so limits the flexibility and availability of services from the service consumer perspective. Moreover, different composition approaches will also affect the other upstream

and downstream software engineering activities, which in turn incur significant cost for both service providers and consumers in a multiplying effect.

The same arguments regarding composition above also hold when other downstream activities such as deployment and run-time management are concerned. Since standards corresponding to these activities have yet to be defined and developed, they can only be conducted in a platform- or vender-specific ways. Consider a typical situation where a developer chooses the popular Axis Web service container [3] as the target platform for service development and deployment. Within the Axis development environment, the developer needs to configure and deploy individual services using XML configuration files defined by Axis. Since Axis has no notion of service composition, that is, mechanisms to allow the developer to combine services at deployment time, even when all the services are developed in house specifically targeting Axis, the developer either has to define his/her own composition mechanism and embeds it into the service implementation, or has to rely on other composition standards and commercial packages such as WS-BPEL.

To fully exploit the potential of SOC, we believe the underlying SOA not only should center on service composition, but also should provide common standards and conventions that can cover most, if not all important software engineering activities for the development of service-oriented applications. One should be able to streamline these activities in a platform-neutral manner as much as possible without being locked down to specific implementation technologies.

In addition, we also believe that future SOC should target not only skilled developers but also ordinary users. In other words, the future Internet is not necessary a simple producer-consumer platform where software developers and end users play their assigned producer and consumer roles, respectively. Instead, users with different skills and expertise can contribute to the global SOC environment differently. To state more generally, we envision the next-generation, service-oriented Internet as a *universal, virtual workspace* (UVW), in which people create and share arbitrary resources – not just simple Web contents but also more complex software artifacts – on the Internet and make them accessible to others.

In this thesis, we are concerned with requirements and challenges towards the ultimate UVW goal, which demands not only robust infrastructure support for the development,

deployment, and assembly of diverse software artifacts, but also effective strategies and mechanisms that can handle the implied complexity.

We propose a more comprehensive SOA to overcome some of the obstacles mentioned above. For example, our SOA standardizes the role of *service containers*, which are responsible of governing the definition, instantiation, customization, composition, and other lifecycle activities for the services they host in a coordinated manner. In addition, our SOA includes a canonical, XML-based description format that serves as the basis for service description, discovery, and composition. In particular, service containers can instantiate run-time services and manage the inter-connections among them based on their corresponding descriptions. Finally, our SOA also include the role of *resource repositories* where service containers can upload and/or download *arbitrary* types of resources, including their descriptions.

In addition to the fundamental SOA, other higher-level facilities and applications are also being developed. For example, current approach to service composition is primarily interface-based. There is no additional mechanism that can assure the correctness and quality of individual services. To address this issue, we develop a testing-based framework on top of our SOA. The framework uses standardized test scripts as supplement information to existing, syntax-based service interface description, to permit service validation at semantics level.

The rest of the thesis is organized as follows. In chapter 2 we describe our approach towards the UVW goal. In chapter 3 we describe our framework for service description and composition. In chapter 4 we describe a testing-based approach to automatic service composition and validation. In chapter 5 we describe further implementation details using a motivating example. In chapter 6 we discuss some of the design considerations and related work, and finally conclude this thesis in chapter 7.

Chapter 2. Universal Virtual Workspace

The vision of UVW is depicted in Figure 1, in which people collaborate by sharing information and software artifacts. Like WWW, the UVW is based on a universal *resource space* in which resources of arbitrary types are identifiable through URIs. Because resources may also embed references to other resources, the resource space in fact forms a globally interconnected network, in a way mimicking the hyperlinked Web pages and multimedia resources in WWW. In addition, UVW also includes the notion of *resource deployment* such that end users can download resources from repositories into their local machines respectively, obtain updates afterwards, or even "upload" their changes.

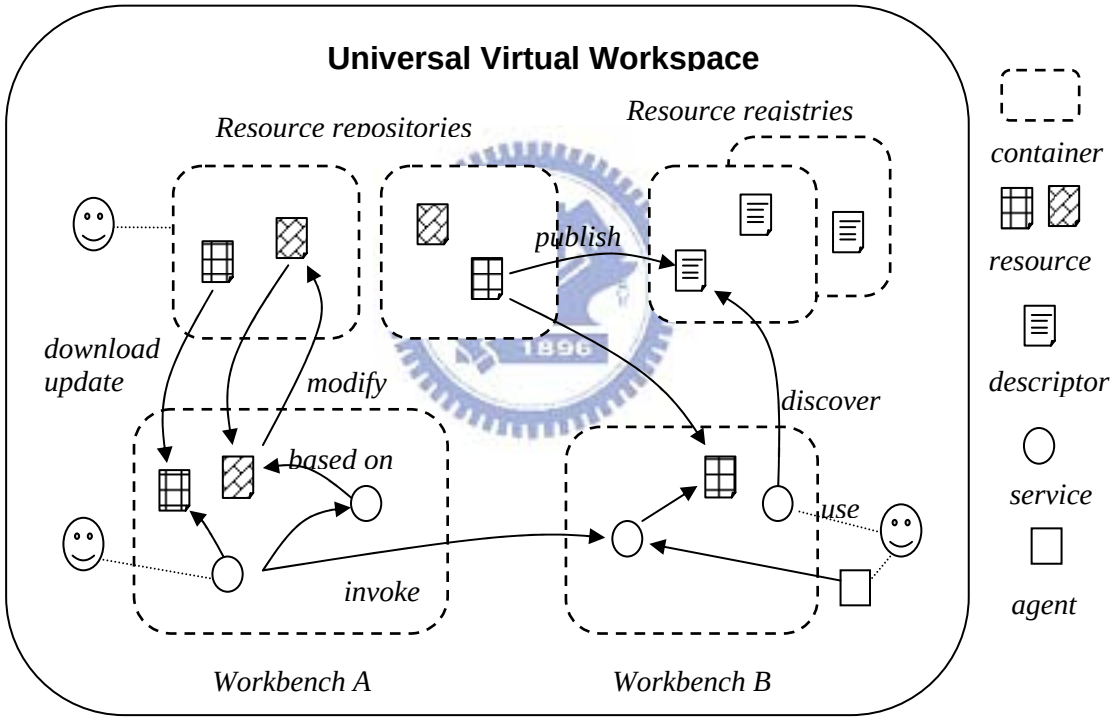

Figure 1. Universal virtual workspace

Because some of the resources are software artifacts that are executable themselves or are components of other executable artifacts, end users can also create application instances and interact with them accordingly. Therefore, the UVW also entails a universal service *composition and execution* platform. To realize such an execution platform, a more detailed SOA, as depicted in Figure 2, is proposed. As shown in the figure, on top of the "persistent" resource space is the dynamic service space that comprises mutually interacting run-time

objects, called *services*, which perform tasks upon requests. Both resources and services are managed by *containers*.

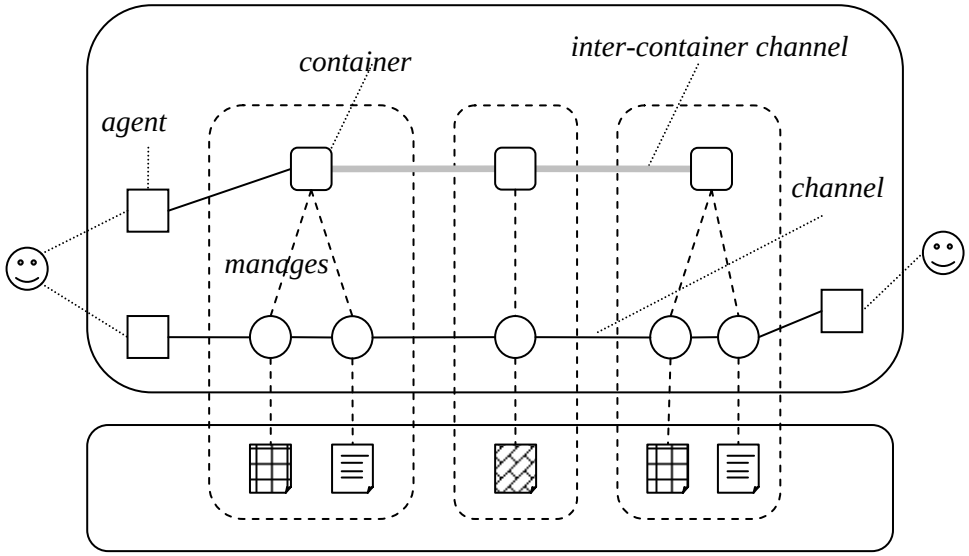

Figure 2. A service-oriented architecture

More importantly, as also indicated in Figure 2, each service is associated with exactly one resource, where the association is *interpreted and maintained by the managing container*. (Accordingly, we may use the term service or resource interchangeably in what follows.) A service can access another service directly if both are in the same memory space, or via some kind of communication channel. In either case, the container is responsible of establishing the suitable channel between the two services, rather than letting them establish links on their own.

Containers can be implemented differently, and they join and leave the global service space continuously. Some containers are long running at server side accepting requests; others may be transient at client side interacting with users. A container may offer different levels of management capabilities to different classes of users – although there are basic operations all containers need to support. Note that although containers are also services, their instantiation and management are system dependent.

A container can also serve as a resource repository as well as a resource registry. In addition, a container may also present itself to the user as a *workbench* through which the user can access and assemble services. The services being assembled may be locally hosted by the workbench, or they may reside in remote containers and accessed through network

communication. The locally hosted services are instantiated by first downloading their associated resources from some remote containers into the local container, and then let the local container interpret these resources correspondingly.

Note that there are also software applications that are not managed by containers but serve as intermediaries between end users and services. We refer *agents* to these software entities.

Chapter 3. Ontology-based Resource Description and Composition

An important design objective of our SOA is to provide a resource description framework in which resources can be defined or annotated using canonical descriptors that can also be tailored for different application domains. In what follows we refer to these resource descriptors as *metaphors*. Unlike existing approaches to resource description such as RDF and OWL that are commonly used in the Semantic Web community, our description framework imposes only syntactical constraints on the contents of and the inter-relations among metaphors, but leave their interpretation to the containers. Furthermore, our SOA also requires that service composition is achieved through syntactically valid metaphor composition (see below). In this chapter we describe our approach to resource description and composition using a simplified example, which can illustrate most of the characterizing features of our SOA.

Consider a simplified scenario in which an organization proposes an e-book ontology that describes relevant terminology, data types, software interfaces, and so on, with the goal to enable a service market where end users can assemble their own book readers using components available in the UVW, browse book databases provided by others, or even create their personalized book databases. To simplify further, assume the e-book domain contains only three types of entities, namely, book viewers, book entry browsers, and book databases. Figure 3 depicts an example that includes three containers, i.e. *User*, *Book.org*, and *Book.com*, each maintaining different types of resources in the e-book domain. As indicated in the figure, *Book.org* maintains the three fundamental resource types, while *Book.com* provides a database implementation conforming to the book database type. Finally, *User* manages resources implementing those viewers, as well as a top-level GUI frame.

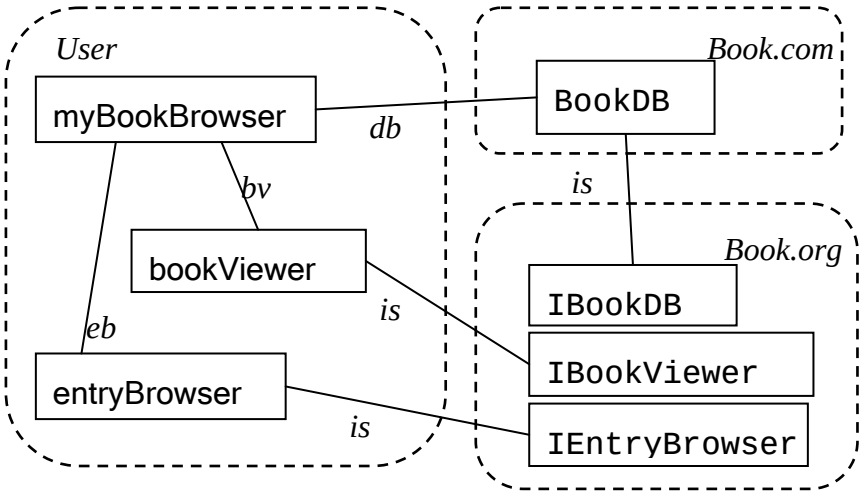

Figure 3. An on-line book reader scenario

Naturally, we choose a subset of well-formed XML documents as metaphors. In general, a metaphor can contain arbitrary contents without restrictions, because the semantics is defined *elsewhere* by its creator. However, we allocate a special namespace with some reserved "keywords" that can be embedded inside metaphors to constrain their *structures* without semantic implications. For simplicity, in what follows the prefix "m" is assumed to be bound to such a meta-level namespace.

To illustrate the syntax of metaphors, consider the metaphors denoting `IBookDB`, `BookDB`, `myBookBrowser`, and `bookViewer` given below:

```
<IBookDB> <!-- //Book.org/ -->
  <m:is m:uri="//ws.org/WebService"/>
  <wsdl m:uri="./bookdb.wsdl"/>
</BookDB>

<BookDB> <!-- in Book.com -->
  <m:is m:uri="//Book.org/IBookDB"/>
  <access url="http://Book.com/bookdb"/>
</BookDB>

<myBookBrowser> <!-- //User/ -->
  <m:is m:uri="//meta/java/Object"/>
  <class name="my.BookBrowser"/>
  <db m:uri="//Book.com/BookDB"/>
  <bv m:uri="./bookViewer"/>
  <eb m:uri="./entBrowser"/>
</myBookBrowser>
<bookViewer> <!-- //User/ -->
```

```
  <m:is m:uri="//Book.org/IBookViewer"/>
  <m:is m:uri="//meta/java/Object"/>
  <class name="my.BookViewer"/>
</bookViewer>
```

In the example metaphors above, the bold-faced "keywords" are meta-level constructs used to describe relations among resources. In particular, **m:is** indicates the *is-a* relation between the current resource and the target resource identified by the **m:uri** attribute. There are also other basic constructs used to constrain the metaphor contents of and relations among resources. For example, the metaphor denoting the resource at "//ws.org/WebService" is sketched below:

```
<WebService> <!-- //ws.org/ -->
  <m:rel name="wsdl" m:uri="//meta/File"/>
  <m:elem tag="url">
    <m:attr name="url" type="URL"/>
  </m:elem>
</WebService>
```

In the example above, **m:rel** imposes a constraint that any resource conforming to `WebService` (via is-a relation) should have a "wsdl" relation with a WSDL document, as indicated by the `<wsdl>` element of `IBookDB`. Similarly, **m:elem** and **m:attr** place some constraints over the structure and content of the metaphor.

Specifically, a metaphor is associated with a globally unique URI and can contain several URI-valued **m:uri** attributes within. These relative or absolute URIs serve as references to other *existing* resource. **m:rel** is a top-level element of a metaphor to constrain the *relations* from the metaphor to others. Syntactically, it states that the metaphor may have a certain *number* of top-level elements with specific *tag* prescribed by the "card" and "tag" attributes of the **m:rel** attribute, respectively. Furthermore, its **m:uri** attribute also constrains the *type* of the metaphor it can be related to (see below). **m:elem** and **m:attr** elements constrain the "data" part of metaphors with intuitive meaning. Finally, **m:is** is a top-level element of a metaphor used to indicate that the metaphor should *inherit* everything from the metaphor referenced to by its **m:uri** attribute and *conform* to its constraints. For brevity, we also call the latter the *type* of the former. Metaphors may be *invalid* if they contain constraints that conflict with each other.

In general, metaphors can be created to annotate other actual resources, or they can be

resources themselves (i.e. self-describing). In fact, it is common to create metaphors to capture high-level concepts without explicitly defining their semantics, and use them immediately to describe other existing resources. In other words, metaphors together form a user-definable ontology, or more precisely, multiple ontologies within the UVW. Because of the integral deployment mechanism in our SOA, it is straightforward for people to publish their own ontologies for the problem domains that concern them; public ontologies can also be created and maintained on a community basis so that containers or service developers joining the same community can interoperate and communicate.

The examples above also highlight our approach to service composition, which is achieved through metaphor composition, provided that all syntactical constraints among metaphors are satisfied, and the semantic implications are agreed upon by the participating containers. Take `myBookBrowser` as an example; it suggests that the container should instantiate a service using the class named "`my.book.BookBrowser`". In addition, when the service needs to access its component services, it can ask the container by supplying corresponding relation names (db, bv, eb) without worrying about how they are instantiated.

Because metaphor syntax and semantics are domain- and container-specific, there is no restriction about how services should be instantiated, assembled, or managed. This feature is essential to fulfill the requirement that the platform can support arbitrary domains and users of varying background suitably. However, reusability and interoperability can be compromised when, for example, similar but incompatible languages are created.

What we want is a virtual service assembly platform that promotes reuse. It is "recommended" that the *manager-worker separation principle* is enforced such that services are like workers who concentrate only on what they are designed for without worrying about how their supporting "colleagues" are created and accessed – all these tasks are the responsibility of the container, thus making services more focused and reusable. In the example above, all services should not hardcode the knowledge of where they or their colleagues are in their computation logic.

With the principle in mind, service composition can be encapsulated into *templates* for reuse. Service composition can further be encapsulated into *templates* for reuse. Simply speaking, templates are "unfinished" metaphors with constraints to be fulfilled; hence their forms may range from simple to complex. Skilled users may create complex templates

combining services with sophisticated gluing logic but with straightforward “parameters” for end users to fill. Different containers can offer different, possibly unique templates for clients. As a simple example, to help promoting the database service, *Book.com* can also provide a GUI frame and a template for others to download:

```
<BookBrowser> <!-- //Book.com/ -->
  <m:is m:uri="//meta/java/Object”/>
  <class name=”com.Book.BookBrowser”/>
  <db m:uri="./BookDB"/>
  <m:rel name=”bv” m:uri="//meta/java/Object"/>
  <m:rel name=”eb” m:uri="//meta/java/Object"/>
</BookBrowser>
```

Chapter 4. Towards Automatic Service Composition and Testing

The SOA described thus far can already enable a generic and flexible platform for service deployment and composition, but is nevertheless too developer-oriented. To make it possible for ordinary people to assemble services into quality systems, additional supports are needed. In this chapter we describe one such support that is simple yet effective. The idea is that, in addition to core data and service types, the domain initiator can also publish test scripts as supplement information. When a community forms in which people agree upon the syntax and semantics of these supplement information, it becomes possible to perform composition and validation automatically with proper tool assistance.

In our testing framework, test scripts are divided into *general* and *template-specific* ones. The former are for unit and integration testing against public interfaces, while the latter may contain implementation-specific information and are mainly for validating template instances. In either case, a test script includes necessary information regarding the interfaces or templates it is designed for.

Figure 4 illustrates our testing framework schematically. As the figure shows, the user is investigating candidate services of types A and B, respectively. First, with tool assistance from the workbench, the user obtains candidate services a_1, a_2, b_1 and b_2 with their corresponding service types, respectively. Similarly, general test scripts that are associated with A and B can also be gathered automatically (i.e. t_1, t_2, t_3). With the candidate services and test scripts at hand, a test plan can be synthesized and executed automatically which tests various combinations of services to see whether they are functioning correctly or can work with each other properly. The kinds of tests performed and the degrees of thoroughness depend on the test scripts and can vary dramatically without limit.

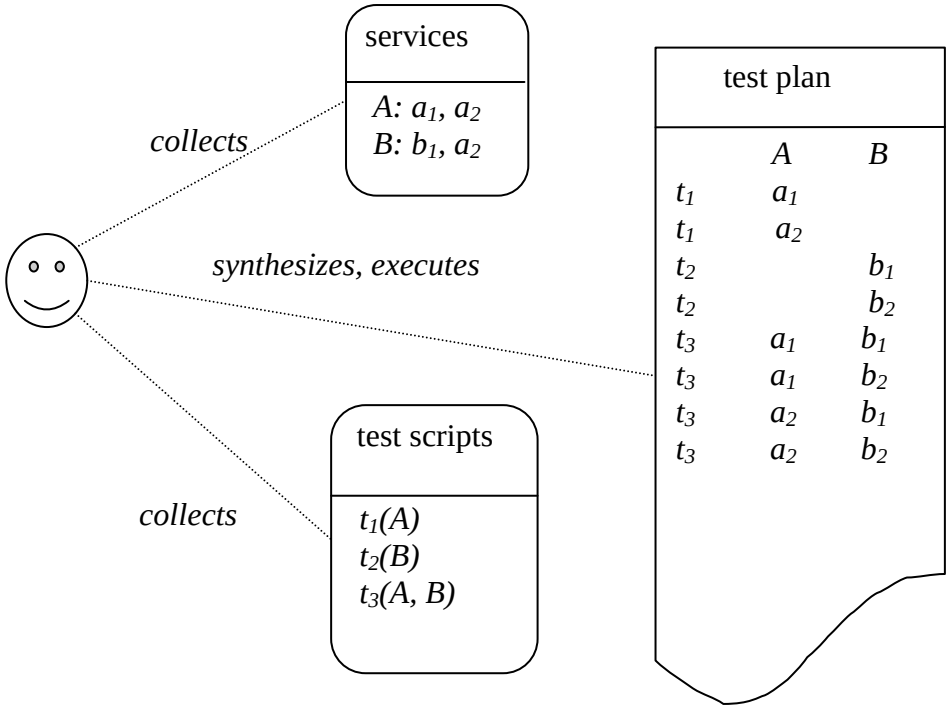

Figure 4. A simple testing scenario

Intuitively, template-specific test scripts are simply those against instances of specific templates; therefore they may exploit implementation details specific to the template. Still, the same unit testing and integration testing that involve general test scripts as outlined previously are still needed for the parameter services of the template instance.

Consider the same e-book example described previously. In addition to the template for instantiating the (proprietary) book viewer, *Book.com* can also publish additional test scripts which we simplify below:

```
<BookBrowserTest> <!--//Book.com/-->
 <m:is m:uri="//Test.org/Test"/>
 <needs name="bookViewer" m:uri="//Book.org/IBookViewer"/>
 <needs name="entryBrowser" m:uri="//Book.org/IEntryBrowser"/>
 <bookBrowser m:uri="//Book.com/BookBrowser"/>
 <m:is m:uri="//meta/java/Object"/>
 <class name="com.Book.BrowserTest"/>
</BookBrowserTest>
```

These test scripts assume that *Book.com*'s own book database is used, but are still generic with respect to which book viewer and entry browser are used. With this information available, our testing framework can search for all conforming book viewers and entry

browsers available locally and automatically create a test plan that cover all possible combinations for the book browser template.

In this example, generic test scripts are still needed. For example, standard or implementation-specific scripts for *unit testing* that test individual services, such as the database and the book viewer above, can all be gathered automatically and included in the test plan. This is particularly useful for users who want to create their own applications or templates without relying on pre-existing templates. Because of our syntax-based description framework, it is not difficult to develop more user-friendly tools that help users create syntactically correct metaphors.

Chapter 5. Implementation

Based on service-oriented architecture described previously, we have developed a framework that permits automatic service composition and verification through testing. The functions of the framework include browsing and downloading existing components, and combining components to form a desired application. In addition, it supports automatic testing by automatically synthesizing test plans based on user-provided test cases. Figure 5 below shows the screen shot of our framework:

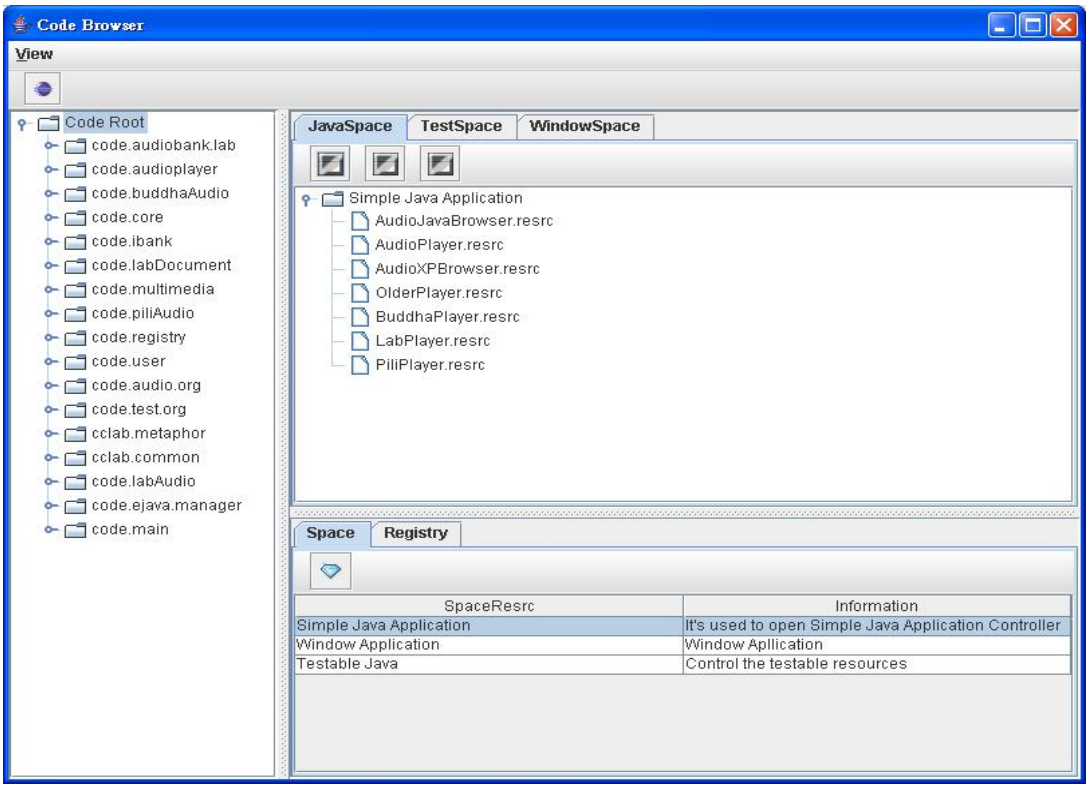

Figure 5. Screen shot of our framework

To illustrate various functionality of our framework we will use a motivating example below. Consider an *audioplayer* application domain that is similar to the e-book application domain described in previous chapters.

First, to establish the audioplayer application domain, one need to determine what kinds of artifacts should actually be deployed. In this example, it is assumed that software artifacts are Java classes; hence they assume the existence of a JVM and some core libraries such as Swing library on the user's machine. Unlike other deployment standards such as OSGi, our

framework does not limit the types of artifacts to be deployed. For example, it is possible to design different application domains which include Web pages or other multimedia resources. To distinguish different classes of artifacts recognized and managed by our framework, the domain initiator need to indicate which *Space* the domain artifacts belong to. A space in our framework is basically an artifact manager that recognizes a special class of artifacts and manages them accordingly. Figure 6 below shows a list of spaces that our framework recognizes.

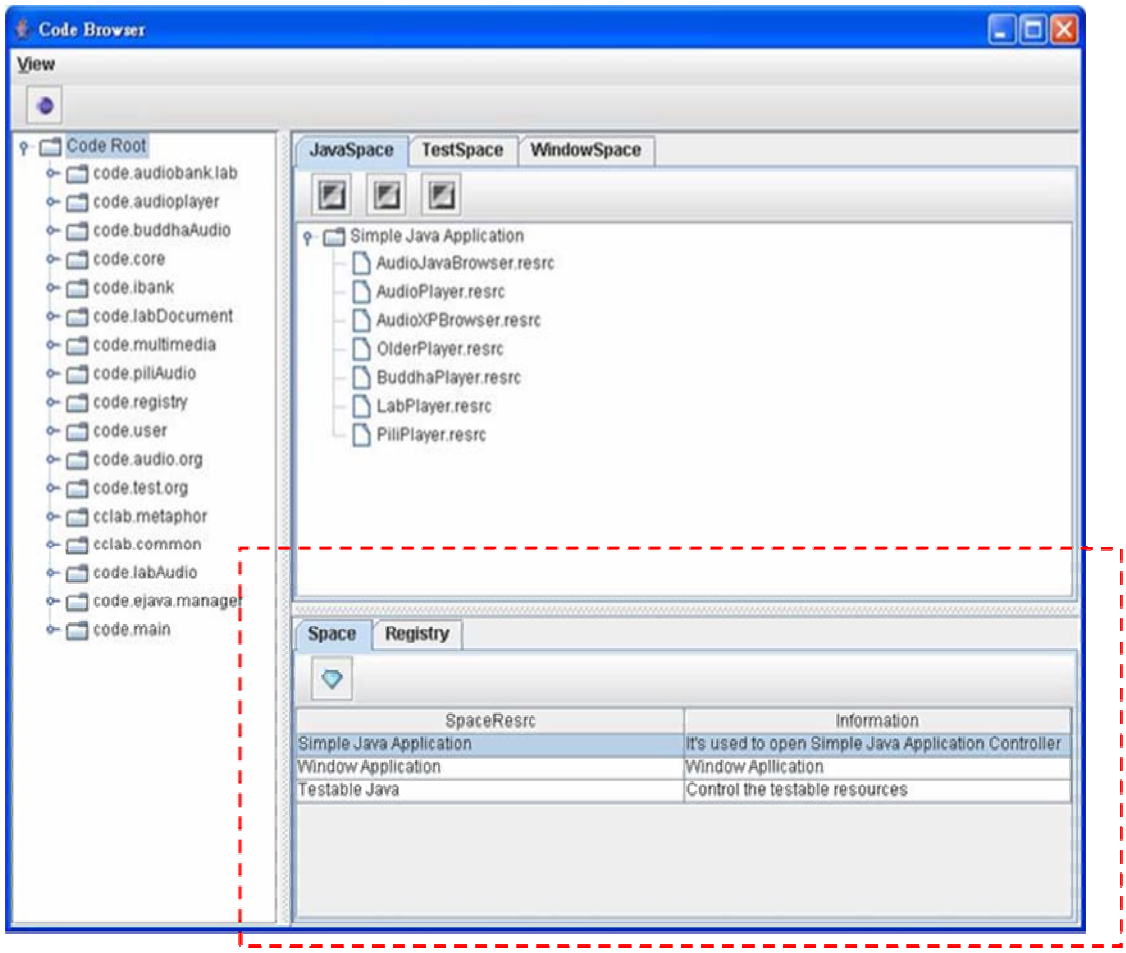

Figure 6. Space manager

For our audio player example, we assume a simple Java space where classes and other relevant resources such as audio files, icons, and so on are recognized. Figure 7 depicts a specific audio player instance consisting of three major entities: Database, Player and Browser. Database stores the music resources; Player plays the audio resources stored in Database; Browser is the main user interface providing usual audio player operations.

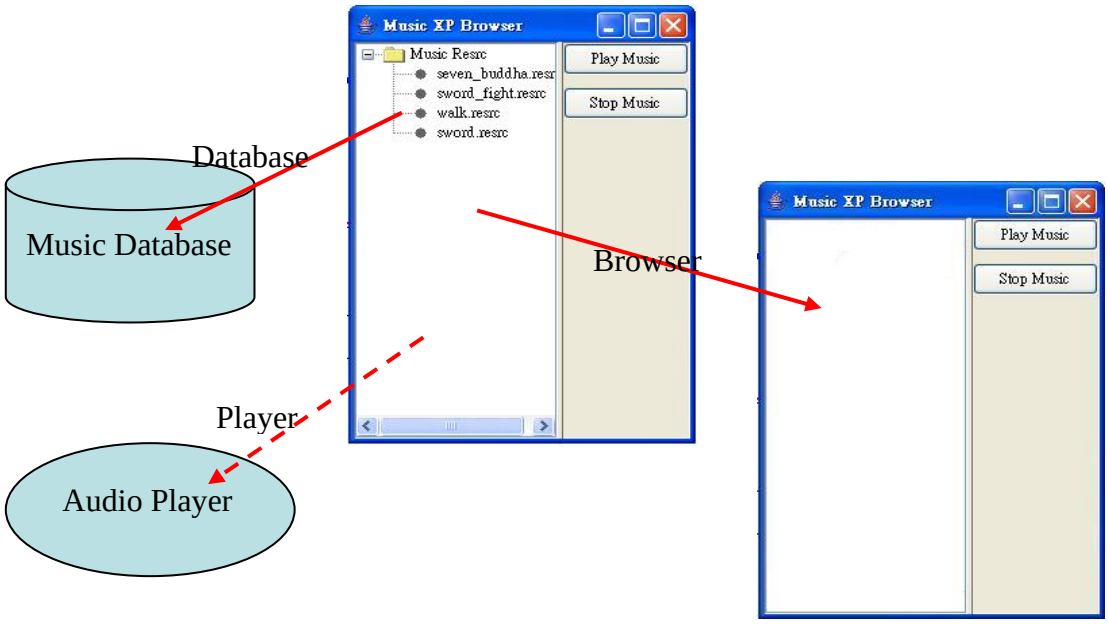

Figure 7. An audioplayer application example

Figure 8 illustrates a component composition by an average user using a template. According to the template that may be provided by some organizations or service providers, the user can simply choose the components whose types are constrained by the template. As Figure 8 shows, there is an Aduio.org organization, and it defines three public interfaces for implementation. There are three companies: Company A provides an *AudioJavaBrowser* component implementing the interface *IAudioBrowser*, and *BuddhaDB* implementing the interface *IAudioDB*. Similarly, Company B provides *AudioXDBrowser* implementing the interface *IAudioBrowser*, as well as *LabDB* implementing the interface *IAudioDB*. Finally, Company C just provides an *AudioPlayer* component that implements the interface *IAudioPlayer*.

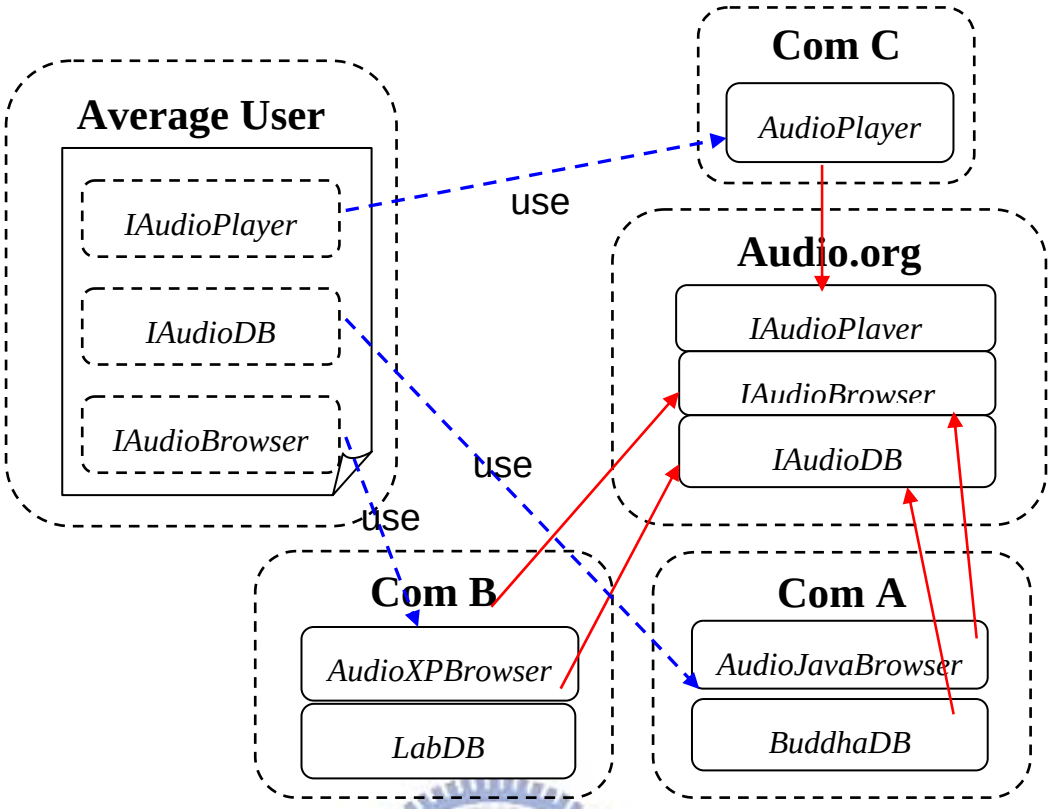

Figure 8. An audioplayer template for average user

The actual metaphors that describe the inter-relations among these different types of artifacts are explained below. First, we show the metaphors that represent the public interfaces below:

```
<IAudioPlayer m:uri="//Audio.org/IAudioPlayer">
  <doc uri="http://Audio.org/IAudioPlayer.html"/>
</IAudioPlayer>

<IAudioBrowser m:uri="//Audio.org/IAudioBrowser">
  <doc uri="http://Audio.org/IAudioBrowser.html"/>
</IAudioBrowser>

<IAudioDB m:uri="//Audio.org/IAudioDB">
  <doc uri="http://Audio.org/IAudioDB.html"/>
</IAudioDB>
```

In the metaphors above, the **m:uri** attributes indicate the global URIs of the public interfaces, respectively. What these interfaces imply are outside the scope of metaphors. In this case, their semantics should be consulted based on their corresponding documents indicated in the doc elements, respectively.

With these public interfaces, templates for specific audio players can be created. Consider the template below:

```
<AudioPlayerTmpl>
 <m:rel name=”player” m:uri=”//Audio.org/IAudioPlayer”/>
 <m:rel name=”browser” m:uri=”//Audio.org/IAudioBrowser”/>
 <m:rel name=”db” m:uri=”//Audio.org/IAudioDB”/>
 <main class="code.audioplayer.AudioPlayerFrame"/>
</AudioPlayerTmpl>
```

The template suggests that we can simply choose suitable implementations and plug them together to make up an audio player. The “class” attribute of the main element in the template indicates that there exists a Java class implementation (that is, `code.audioplayer.AudioPlayerFrame`) in the user’s environment such that for a given instantiation of the template, a corresponding Java object can be instantiated, which will then contact other component services supplied in the template instance. For example, the template instance below shows that a service (instantiated based on) *AudioXPBrowser* from Com B, *BuddhaDB* from Com A, and *AudioPlayer* from Com C are used to fulfill the required parameters of the template: *IAduioBrowser*, *IAudioDB* and *IAudioPlaye*.

```
<myAudioPlayerFrame>
 <m:is m:uri=”./AudioPlayerTmpl”/>
 <player m:uri=”./myAudioPlayer”/>
 <browser m:uri=”./myAudioXPBrowser”/>
 <db m:uri=”./buddhaAudioDB”/>
</myAudioPlayerFrame>

<myAudioPlayer>
 <m:is m:uri="//Audio.org/IAudioPlayer"/>
 <main class="code.audioplayer.AudioPlayer"/>
    ...
</AudioPlayer>

<myAudioXPBrowser>
 <m:is m:uri="//Audio.org/IAudioBrowser"/>
 <main class="code.audioplayer.ui.AudioXPBrowser"/>
      ...
</myAudioXPBrowser>

<buddhaAudioDB>
 <m:is m:uri="//Audio.org/IAudioDB"/>
 <main path="/code.buddhaAudio/resource"/>
     ...
</buddhaAudioDB>
```

Consider another case where the user is skilled enough to develop their own components in Java, and just uses one or more external components from other companies. For example, as shown in Figure 9, the user develops *MyBrowser* and *MyPlayer*, and chooses *BuddhaDB* from Company A that implements the *IAudioDB* interface.

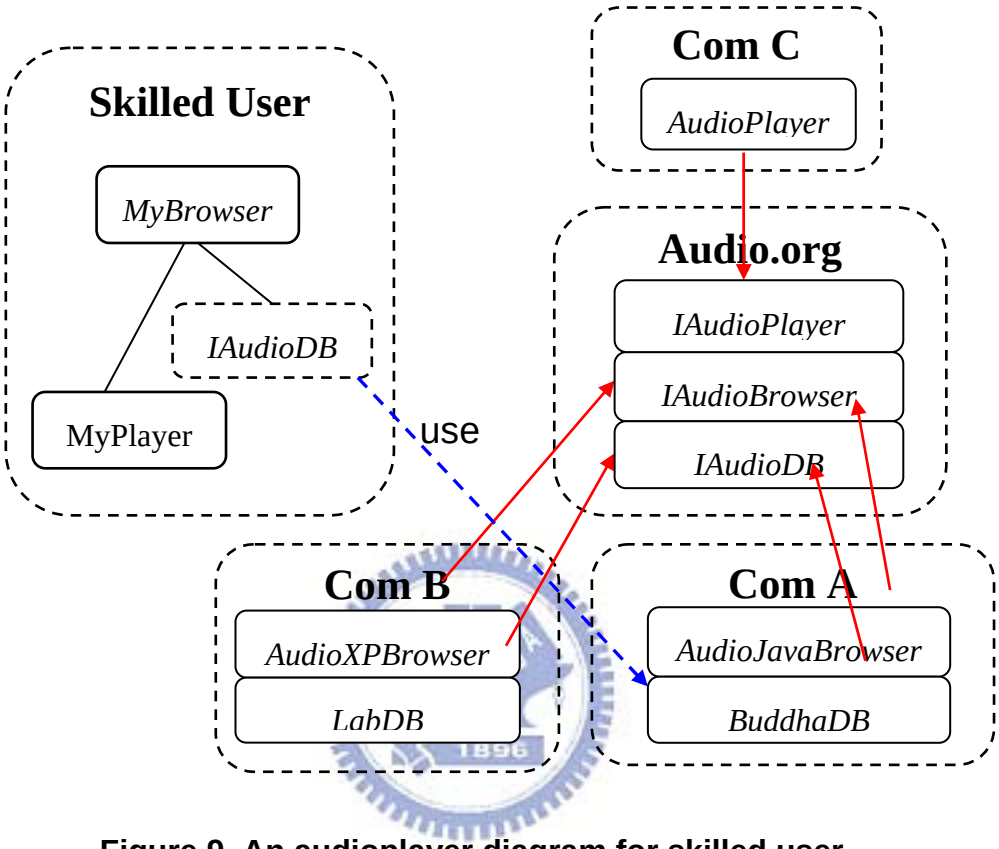

Figure 9. An audioplayer diagram for skilled user

After composing components, we have to test the composition. As mentioned previously, automatic testing can also be supported when creating a particular application domain so that it becomes possible to verify the correctness and conformance of individual components as well as their composition. Below we use the same audio player application domain to illustrate our testing framework implementation.

Consider the testing related concepts depicted in Figure 10 that are part of the audio player domain.

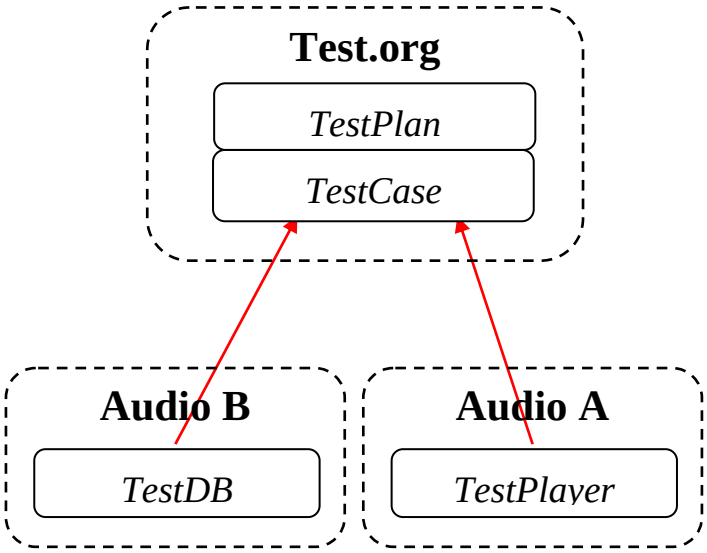

Figure 10. An audioplayer testing diagram

In the figure, there is an organization Test.org which defines the standard concepts *TestPlan* and *TestCase* related to testing. There are also two companies Audio A and Audio B who participate in the domain and provide test cases *TestPlayer* and *TestDB*, respectively, that implement the interface *TestCase* from Test.org. Furthermore, suppose by implementing the *TestCase* interface it means that the test case can be used to test individual or some combination of components whose types are indicated in the "SUT" (System Under Test) attributes of its "needs" elements. For example, the TestPlayer test case from Audio A shown below indicates that it can be used to test components of IAudioPlayer type:

```
<TestPlayer m:uri="//AudioA/TestPlayer">
  <m:is m:uri="//Test.org/TestCase"/>
  <java class="AudioA.test.TestPlayer"/>
  <needs SUT="//Audio.org/IAudioPlayer" as="player"/>
  ...
</TestPlayer>
```

It is not difficult to see that the corresponding metaphor denoting *TestCase* looks like:

```
<TestCase m:uri="//Test.org/TestCase">
  <m:elem tag="java">
    <m:attr name="class" type="String"/>
  </m:elem>
  <m:elem tag="needs" card="*">
    <m:attr name="SUT" type="URI"/>
    <m:attr name="as" type="String"/>
  </m:elem>
  ...
```

```
</TestCase>
```

Note that unlike the previous examples where the requirement of Java implementation are not implied, here to automate testing, *TestCase* also imposes the requirement that any test case should be implemented as a Java class, so that related test cases designed for specific SUTs can be gathered automatically by the framework and corresponding test plans can be synthesized and executed automatically. Below shows a simplified test case implementation in Java that helps illustrate our approach.

```
public class TestPlayer extends TestCase {
  public void runTest() {
    testAuthor();
  }
  public void testAuthor() {
    IAudioPlayer player = getPlayer();
    assertNotNull(player.getName());
  }
  private IAudioPlayer getPlayer() {
    return (IAudioPlayer)getContext("player");
  }
}
```

The Java implementation above also indicates that when a test case is executed, it will obtain the SUT it runs against from its execution context, by supplying a pre-defined name (i.e. “player”) that is also specified in the TestPlayer metaphor.

To perform testing, test plans need to be created first. A test plan is essentially a file describing the set of components and different but feasible combinations among them for testing. Because the number of combinations can be quite large, our framework can assist user to create test plans based on the components he/she is working on. Below shows an example test plan using the audio player example above:

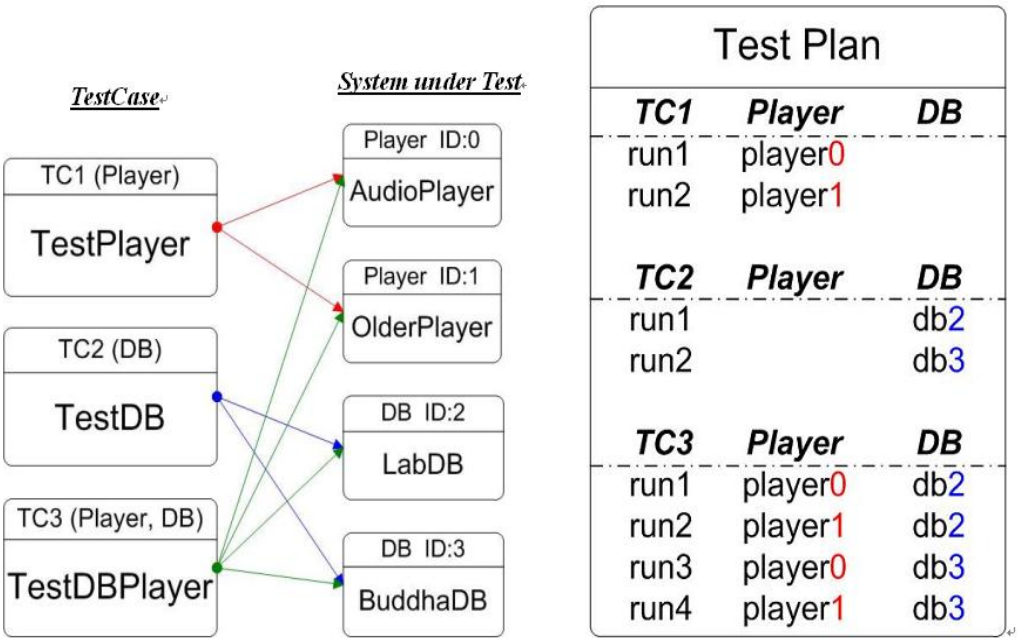

Figure 11 TestPlan diagram

In this example, there are three test cases: *TestPlayer*, *TestDB* and *TestDBPlayer*, and four SUTs: *AudioPlayer* with ID 0, *OlderPlayer* with ID 1, *LabDB* with ID 2 and *BuddhaDB* with ID 3. *TestPlayer* just needs an *IAudioPlayer* type SUT, and it will test the players with ID 0 and 1. The setup is similar for *TestDB*. However, *TestDBPlayer* needs two kinds of SUTs, i.e. *IAdudioPlayer* and *IAudioDB*, so it needs to consider all combinations among available SUTs. A possible test plan is shown below, which depicts the four SUTs with their corresponding IDs: *AudioPlayer*, *OlderPlayer*, *BuddhaDB* and *LabDB*.

```
<testplan>
 <suts>
  <sut id="0" m:uri="/local/java/AudioPlayer"/>
  <sut id="1" m:uri="/local/java/OlderPlayer"/>
  <sut id="2" m:uri="/local/java/BuddhaDB/"/>
  <sut id="3" m:uri="/local/java/LabDB/"/>
 </suts>
  ...
</testplan>
```

The produced test plan is shown below:

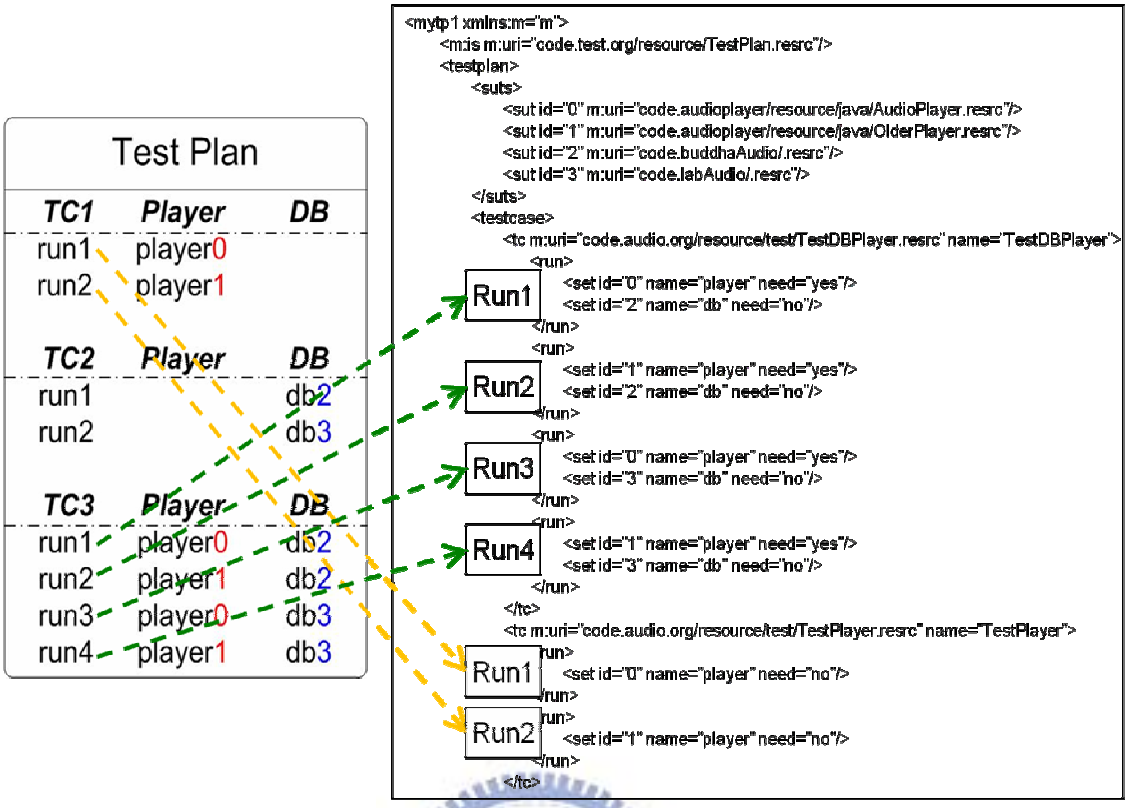

Figure 12 An example test plan

As an example, the test plan above indicates that the testcase *TestDBPlayer* has to run four times. For example, the run1 in *TestDBPlayer* will use the player with ID 0 and the database with ID 2. The output after executing the test plan is given below:

```
Test第1個TestCase with name:TestDB
  此TestCase下第 1 種combination of suts
  sut: name: db with ID: 3
  此Test是否有failure?: 0
  此Test是否成功?: 成功

  此TestCase下第 2 種combination of suts
  sut: name: db with ID: 2
  此Test是否有failure?: 0
  此Test是否成功?: 成功

  ...
  ...
```

```
Test第2個TestCase with name: TestPlayer
  此TestCase下第 1 種combination of suts
  sut: name: player with ID: 1
  此Test是否有failure?: 1
  此Test是否成功?: 失敗

  此TestCase下第 2 種combination of suts
  sut: name: player with ID: 0
  此Test是否有failure?: 0
  此Test是否成功?: 成功
```

Figure 13. Results of a test plan execution

Chapter 6. Discussion and Related Work

We have implemented the SOA and the testing framework based on the popular Eclipse platform [4]. Here we summarize some of the important features of our implementation without going further into details:

- All resources, including metaphors, are managed as files and organized as Eclipse-managed projects. It is possible that some metaphors may be invalid at a given point it time, although the user can perform various consistency checks periodically.
- Deployment is achieved through Eclipse's built-in version control system (i.e. CVS). Because public resource registries and repositories are also projects downloaded into user's workspace, our SOA does not require additional communication protocols for service discovery and deployment.
- Except the metaphor mechanism and file-based storage for resources, the SOA is generic with respect to allowable resource types. For example, Java-based and C-based software artifacts can co-exist within the same SOA, so are two different testing frameworks for the same types of Java-based artifacts.
- Our testing framework implements both Java-based and scenario-based test scripts. Java-based test scripts are implemented as JUnit test cases plus the associated metaphors describing their required service types to facilitate automatic testing. As discussed in the previous chapter, some test scripts can be generic and considered part of the public contract that service implementation should conform to, or vendor-specific and may be bundled with particular implementations. Scenario-based test scripts are conceptually similar, except that they express the expected input/output or message exchanges using XML, thereby providing a more technology-neutral framework.

The vision of UVW is inspired by the concept of "intercreativity" envisioned by Tim Berners-Lee [5] when architecting WWW [6], where people collaborate by creating and posting Web contents for others to see. This together with the emerging service-oriented computing trend have led us to the conclusion that Internet is transforming into a common

medium for people to participate in, rather than just a consumer-producer platform where most people are restricted to access information and services provided by software developers.

An important reason we believe WWW will be a good model for extension is not just because of its ubiquity, but also because of its architectural simplicity that helped propelled the Web. This is reflected by the service space underpinning our SOA that differentiates our approach from existing ones. One advantage is that our SOA can support diverse application domains and distributed computing technologies on a community basis, which is a fundamental requirement for UVW. As the e-book example suggests, for example, resources available for composition are not limit to Web services, and can also include downloadable Java classes that run locally and interact with end users via GUIs. From this perspective, our SOA combines the concepts from "server-side" distributed computing technologies and the "client-side" deployment frameworks that are common in modern Web browsers or operating systems. Similarly, our SOA can also support "light-weight" application domains such as P2P file sharing that may not require too much infrastructure overhead. Unlike WWW, however, our SOA is more of a distributed computing platform holding software artifacts and services, and permit service composition.

Virtualization has been one of the fundamental principles underpinning computer science and information technology, as seen in many research areas including programming languages, operating systems, etc. Virtualization of distributed, heterogeneous resources is recently re-signified by the grid computing [7, 8, 9] research and closely related peer-to-peer computing. One major goal is to utilize otherwise idle, disparate computing resources by joining them into workhorses that approximate super computers. Furthermore, the concept of virtual organization also stresses that the primary emphasis is on effective utilization of distributed resources across organizational boundaries while respecting the authority and policies of individual organizations. This is what differentiates grid computing from distributed operating systems research.

myGrid [10] is one of the famous open source Grid applications that aim to provide a high-level service-oriented middleware to support in-silico biological experiments. Interestingly, myGrid includes the Taverna workbench as one of its core component, which allows biologists, rather than developers, to create workflows connecting third-party Web services via more intuitive, graph-based user interfaces. The workbench supports individual

scientists by providing personalization facilities related to resource selection, data management and process enactment. Figure 14 illustrates myGrid Taverna workbench. On the top-left, it shows available services for use. User can choose the desired services, add them to the workflow diagram, and set the process between services to compose a workflow with graph-based user interface.

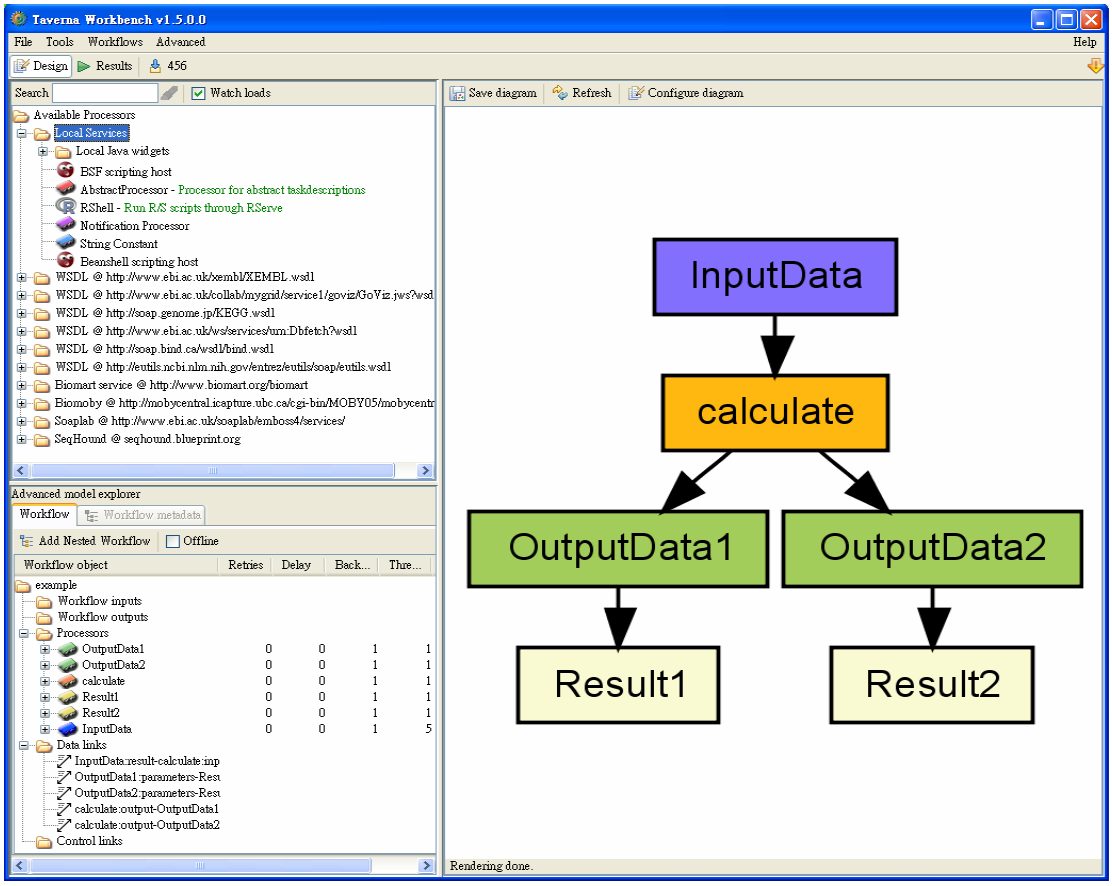

Figure 14. myGrid Taverna workbench

The workflow can be saved as XML-based description, which is illustrated below:

```
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha"
 version="0.2" log="0">
<s:workflowdescription
 lsid="urn:lsid:net.sf.taverna:wfDefinition:544956f5-42dc-47e3-bbca-d4
 ffcec13f0b"author="" title="example"/>
<s:processor name="OutputData1">
 <s:arbitrarywsdl>
 <s:wsdl>http://eutils.ncbi.nlm.nih.gov/entrez/eutils/soap/eutils.wsdl
 </s:wsdl>
  <s:operation>run_eSearch_MS</s:operation>
 </s:arbitrarywsdl>
```

```
</s:processor>
<s:processor name="calculate">
<s:description>RENCI impl for blast service</s:description>
<s:biomobywsdl>
<s:mobyEndpoint>http://mobycentral.icapture.ubc.ca/cgi-bin/MOBY05/moby
 central.pl</s:mobyEndpoint>
<s:serviceName>Blastn</s:serviceName>
<s:authorityName>biomoby.renci.org</s:authorityName>
</s:biomobywsdl>
</s:processor>
 ...
<s:processor name="InputData">
<s:seqhound>
<s:method>SHound3DExists</s:method>
<s:server>seqhound.blueprint.org</s:server>
<s:jseqremserver>skinner.blueprint.org:8080</s:jseqremserver>
<s:path>/cgi-bin/seqrem</s:path>
<s:jseqrempath>/jseqhound/jseqrem</s:jseqrempath>
</s:seqhound>
</s:processor>
<s:link source="InputData:result" sink="calculate:input" />
<s:link source="OutputData1:parameters" sink="Result1:first_url" />
<s:link source="OutputData2:parameters" sink="Result2:sbegin" />
<s:link source="calculate:output" sink="OutputData1:parameters" />
```

Recently, this trend in user-centric, collaborative computing has gained some momentum. Consider the widespread use of Web applications such as blogs and Wikipedia [11]. These applications provide easy-to-use interfaces that allow people to create contents such as opinions and photos for others to see. Equally importantly, they provide storage and content management facilities under the hood. Although the user interfaces are often limited (for ease of use), these applications already provide sufficient functionality people want. As a result, the simplicity helps these applications gain huge user base in a short period of time, which is often attributed by Web 2.0 [12] promoters as *network effect*.

Web 2.0's emphasis on sharing and collaboration among end users, not developers, coincides with our view. On the other hand, the tendency of Web 2.0 application developers to centralize their proprietary implementation behind (high-performance) servers – a key characteristic for Web 2.0 companies to stay ahead – is in contrast to our UVW goal. As a result, Web 2.0 does not consider too much about a common computing infrastructure, or about the assembly of third-party modules, and the issue of software deployment and maintenance are considered irrelevant.

Our approach to describing and interpreting resources via metaphors provides a composition framework that promotes domain-specific, language-based component reuse, in the sense that new types of resources are conceived with corresponding languages defined and interpreters developed. Specifically, in the resource space, new resources can be created for a given domain (generic or domain specific); in this case, custom language syntax can be defined for the customization of a certain class of resource. In the service space, new communities or domains can be created by equipping containers with differentiating interpreters. The access interface to the container, the composition mechanisms, and the corresponding assembly languages are all extensible.

In short, our approach to composition is syntax-based in nature. This is in contrast to most AI-based composition approaches, e.g. the Semantic Web [13, 14] movement and related models such as OWL-S [15], where the main focus is on the development of languages for describing the properties and capabilities of Web services in unambiguous, computer-interpretable form, in order to facilitate automatic reasoning, negotiation, and dynamic integration of Web services.

Our approach also differs from another popular trend, i.e. workflow-based service composition (e.g. WS-BPEL and W3C CDL), which emphasizes on support for cross-organization business processes that are crucial in the coming e-commerce era. Nevertheless, most workflow-based approaches are “server-side” technologies targeting developers and service providers. In contrast, the UVW unifies the server side and the client side, where end users and developers are among the many groups of people in the potentially complex ecosystem. In other words, the UVW can be characterized as a global, integrated development environment supporting “programmers” of various skills and needs.

This software engineering perspective also highlights many important factors that are missing in current AI-based or workflow-based composition approaches. For example, evaluating whether a service performs its duty as it claims to, or managing the versions of component services in a composite service are often beyond the scope of these composition approaches, but are still within our scope.

As another example, one important issue related to service instantiation and management is deployment. Deployment mechanisms are also an important area that receives many research and development efforts recently. Popular Web browsers, for example, often provide

plug-in mechanisms that download executable resources such as Java Applets or Flesh applications and manage them behind the scene for the user. Other deployment mechanisms outside the Web arena are also common; examples include the plug-in architecture of the popular Eclipse IDE, or the Java-based middleware OSGi [16] for component integration, or the Maven project that streamlines software building process by acquiring required libraries across network based on project profiles.

The OSGi Service Platform provides a general-purpose Java framework that supports the deployment of applications (called bundles) and provides the functions to change the composition dynamically without restarting. A bundle comprise of Java classes and other resources such as manifest file describing the information about the bundle to provide functions (services) and to be exported as Java ARchive (JAR) files are the only entities for deploying Java-based applications. A bundle can contain zero or more services and be downloaded, installed, updated and removed in an OSGi environment. A service published in a bundle can be searched and installed in OSGi environment by other bundles for exploiting. Take the Knopflerfish project for example. Knopflerfish is a non-profit organization and aims to develop and distribute easy to use open source implementations of the OSGi frameworks, as well as related build tools and applications. Figure 15 illustrate the Knopflerfish framework with graphical user interface. For example, on the left side of the figure shows the bundles that have been installed and can be started and stopped. User can search and install bundles that have been published and registered to bundle repository from the center part (Bundle Repository). Moreover, users can update and uninstall bundles.

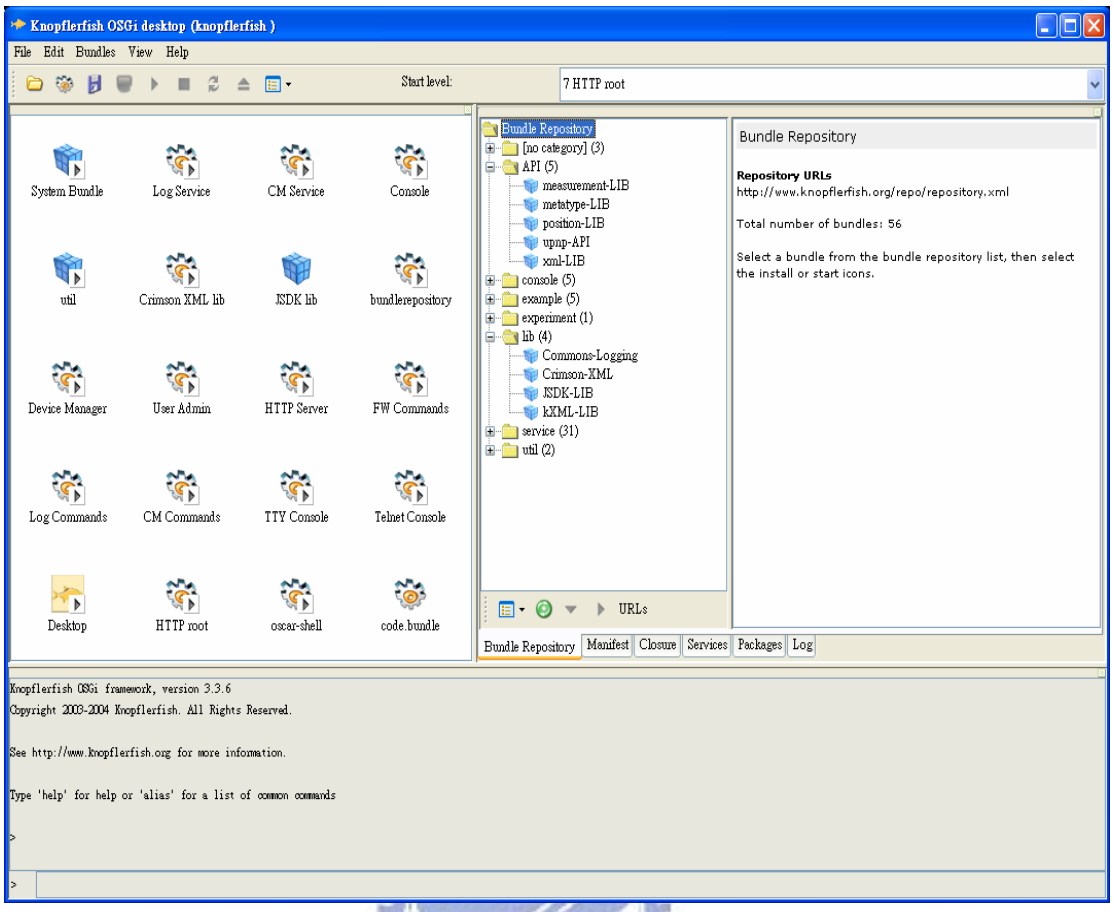

Figure 15. The Knopflerfish OSGi framework

However, these deployment mechanisms focus on managing downloaded modules which often depend on each other in a static, predefined way, and they are not designed for users to assemble novel applications. In other words, deployment mechanisms are currently separated from component or service composition frameworks. In contrast, we are more interested in an environment where both aspects are considered.

Chapter 7. Conclusion and Future Work

We have presented the vision of UVW as a unification of current trends in component-based, service-oriented computing, and user-centric Web 2.0 movement. In realizing the UVW objective, we have also proposed a generic SOA that is

- *resource-oriented*, in a way similar to hyper-linked Web pages and multimedia resources in WWW,
- *ontology-based*, where resources and their composition can be described using user-definable metaphors,
- *a unified deployment, composition, and execution platform*, where the role of containers is made explicit, and
- *user-centric*, targeting groups of users with diverse skills and background.

To facilitate quality service composition, we also proposed a testing-based framework on top of the SOA that can synthesize and execute test plans automatically based on service descriptions and additional test scripts accompanying published service interfaces or implementations.

Of course, there are far more obstacles and challenges than we can address in this thesis in pursuing the UVW goal. One issue is the research and development of satisfactory software engineering environment that even non-technical persons can become productive. Existing development environments are not satisfactory in this aspect, mainly because they rely on the target audience, i.e. developers, to handle the potentially complicated gluing logic among services. Apparently, substantial efforts are needed in order to make the workbench sufficiently intelligent, robust, self-diagnosing, and self-healing.

Also, we leave the security aspect unattended, because the issue is further intensified for every additional requirement we propose for the UVW. In this thesis we focus more on the functional aspects of UVW and the corresponding infrastructure support for flexible composition of distributed, heterogeneous resources. Instead of inventing a security framework ourselves, currently we are working on ways to leverage existing security mechanisms such as those supported by the Globus Toolkit [17].

References

[1] Web Services Activity, http://www.w3.org/2002/ws/

[2] WS-BPEL, OASIS, http://www.oasis-open.org/

[3] Apache, Web Services - Axis, http://ws.apache.org/axis/

[4] The Eclipse platform, http://www.eclipse.org/

[5] T. Berners-Lee, "Realising the Full Potential of the Web", W3C notes, http://www.w3.org/1998/02/Potential.html

[6] W3C, Architecture of the World Wide Web, Volume One, W3C Recommendation, http://www.w3.org/TR/webarch/

[7] J. Kubiatowicz and D. P. Anderson, "The Worldwide Computer: An operating system spanning the Internet would bring the power of millions of the world's Internet-connected PCs to everyone's fingertips", Scientific American, March 2002, pp. 40-47.

[8] L. Smarr and C.E. Smarr, "Metacomputing", Communications of the ACM, 35(6), (1992), pp. 74-84.

[9] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International J. Supercomputer Applications, Vol. 15, No. 3, 2001.

[10] The myGrid Consortium, "myGrid: Middleware for in silico experiments in biology". http://www.mygrid.org.uk/

[11] Wikipedia, http://www.wikipedia.org

[12] T. O'Reilly, "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software", http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

[13] W3C, Semantic Web Working Group. http://www.w3.org/2001/sw/

[14] T. Berners-Lee, J. Hedler, and O. Lassila, "The semantic web", Scientific American, May issue, 2001.

[15] OWL-S, http://www.daml.org/services/owl-s/

[16] The OGSi Alliance, http://www.osgi.org/

[17] Globus Toolkit, The Globus Alliance, http://www.globus.org/