MODEL DRIVEN DEVELOPMENT OF WEB SERVICES AND DYNAMIC WEB SERVICES COMPOSITION

by

FEI CAO

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2005

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **2005**	2. REPORT TYPE	3. DATES COVERED **00-00-2005 to 00-00-2005**
4. TITLE AND SUBTITLE **Model Driven Development of Web Services and Dynamic Web Services Composition**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Alabama at Birmingham,Department of Computer and Information Sciences,Burmingham,AL,35294**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**168**	

ABSTRACT OF DISSERTATION
GRADUATE SCHOOL, UNIVERSITY OF ALABAMA AT BIRMINGHAM

Degree	Ph.D.	Program	Computer and Information Sciences

Name of Candidate Fei Cao

Committee Chair Barrett R. Bryant

Title Model Driven Development and Dynamic Composition of Web Services

Web Services (WS) has emerged as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA), the open standard description language XML and transportation protocol HTML. Therefore, legacy software systems can incorporate WS technology in order to be reused and integrated in a distributed environment across heterogeneous platforms. While WS is gaining its momentum toward wide adoption in the software industry, there are two critical issues yet to be addressed before its power is fully unleashed: 1) the migration of legacy distributed software system toward WS applications; 2) the innovation of new infrastructure, and languages in support of WS application development. The contribution of this dissertation is in these two directions.

First, a comprehensive, systematic, automatable and language neutral approach is presented toward reengineering legacy software systems to WS applications, rather than rewriting the whole legacy software system from scratch in an ad-hoc, language-specific manner. It is noteworthy that this approach is not specific to reengineering WS applications, but can be generalized to reengineering legacy software systems to other applications. Moreover, this approach offers a means for modeling assets exchange in both horizontal direction and vertical direction (along the meta-model stack).

Second, with the dynamic features of both service consumption and provisioning in distributed environment, WS applications are subject to dynamic composition. As such, in a bottom up order, this dissertation presents an infrastructure for dynamic WS composition, and its high-level programming model based on a hybrid of logic programming and imperative programming. In particular, with the logic programming paradigm and the rule inference engine support, not only autonomous composition is achieved, but also WS selection specification can be seamlessly integrated with composition process, which is necessary for achieving customizability, optimization and Quality of Service (QoS) guarantee for dynamic composition.

To Yi,

It’s you that brings love to me so close.

To Mom and Dad,

It’s your love that makes me reach so far.

ACKNOWLEDGMENTS

My deepest gratitude to my advisor, Dr. Barrett Bryant - you are the one that brought me numerous chances, unfaltering patience, meticulous and inspiring advice, and equipped me with the courage to strive through one of the most challenging periods of my life, yet make it one of my most joyful and memorable time as well. Thanks, Dr. Bryant, for ordering me the right track, and letting me enjoy!

I'd like to sincerely thank my committee members. Dr. Jeff Gray, you brought me the new landscape on the horizon, and your vision has always been enlightening. Your dedication to students is the great assets to me and my fellow students. Dr. Rajeev Raje, Dr. Mikhail Auguston, and Ms. Carol Burt, I greatly appreciate the opportunity of being a member of UniFrame research project. Without you, I would not have gained so many valuable experiences. I feel so blessed to have had many precious communications with you. I am indebted to your critical thoughts and perceptive suggestions on my work, which means a lot to me. Dr. Kevin Reilly and Dr. Murat Tanik, I appreciate your great discussions on my work on various occasions.

To my fellow SoftCom-ers, Wei Zhao, Hui Wu, Alex Liu, Xiaoqing Wu, Jing Zhang, Jane Lin, Suman Roychoudhury, Faizan Javed, I cherish our work and fun time together. Thank you for your help and encouragement during past years—you are all my great buddies! Thanks also go to Ms. Kathy Baier and Ms. Janet Sims, who have been so friendly and helpful during my study here.

At this moment, I am particularly grateful to my Mom and Dad from the bottom of my heart—no words are adequate enough to express my appreciation to your love for all my past life. Your love empowers me to steer through ups and downs and makes me come to this point today. Thank you, Mom and Dad. Also to Yi, you are the gift from the heaven that I can never dream of. Your love makes me feel more confident and enlightened for the rest of my life. You are the one that is always in the deepest part of my heart.

Last, I acknowledge the financial support from the Office of Naval Research. The work presented in this dissertation was supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

TABLE OF CONTENTS

TABLE OF CONTENTS

Page

TABLE OF CONTENTS

Page

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF ABBREVIATIONS

AAR	Adaptation Advice Repository
ADL	Aspect Definition Language
API	Application Programming Interface
AOGDM	Aspect-Oriented Generative Domain Modeling
AOM	Adaptive Object Model
AOP	Aspect-Oriented Programming
AOCE	Aspect-Oriented Component Engineering
AUL	Aspect Usage Language
AUS	Aspect Usage Specification
BON	Builder Object Network
BPEL4WS	Business Process Execution Language for Web Services
CBSE	Component-Based Software Engineering
CCM	CORBA Component Model
CDL	Component Description Language
CLR	Common Language Runtime
CIL	Common Intermediate Language
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CTS	Common Type System
DCOM	Distributed Component Object Model

LIST OF ABBREVIATIONS (Continued)

DynaCom	Dynamic Composition
EJB	Enterprise JavaBeans
ER	Entity-Relationship
FCO	First-Class Object
FDA	Feature Diagram Algebra
FDL	Feature Description Language
FODA	Feature-Oriented Domain Analysis
FSM	Finite State Machine
GDM	Generative Domain Model
GFME	Generic Feature Modeling Environment
GME	Generic Modeling Environment
GP	Generative Programming
HTTP	HyperText Transportation Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IIOP	Internet Inter-Orb Protocol
IIS	Internet Information Service
J2EE	Java 2 Enterprise Edition
JIT	Just-In-Time
MDA	Model-Driven Architecture
MDB	Message-Driven Beans
MDD	Model Driven Development

LIST OF ABBREVIATIONS (Continued)

MIC	Model-Integrated Computing
MOF	Meta Object Facility
MS	Microsoft
MTS	Microsoft Transaction Server
OCL	Object Constraint Language
OMG	Object Management Group
ORB	Object Request Broker
P2P	Peer-to-Peer
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service
RMI	Remote Method Invocation
RPC	Remote Process Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
TLG	Two-Level Grammar
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
UMM	Unified Meta-object Model
URL	Universal Resource Locator
UUID	Universally Unique Identifier

LIST OF ABBREVIATIONS (Continued)

VDM	Vienna Development Method
WS	Web Services
WSDL	Web Services Description Language
XMI	XML Metadata Interchange
XML	Exchangeable Markup Language

CHAPTER 1

INTRODUCTION

1.1 Evolution of Component-Based Software Development

Software systems are continually required to address increasing demands of scalability and correctness. To meet these requirements, software development has evolved into a process of reusing existing software assets rather than constructing a new software system completely from scratch [McIlroy,69]. By reducing time-to-market, this approach has improved the economic and productivity factors of software production [Devanbu, 96]. Technically, by separating overall functionality into small units, software reuse also offers a means for better manageability [Brown, 00] and predictability [Hissam, 03] over the constructed software system.

The granularity of software reuse has evolved in tandem with the capabilities of existing programming languages - from functions/procedures found in imperative programming languages, to the object/class mechanisms available in object-oriented programming languages. The current context of software reuse also scales from standalone software development for a single machine, to capabilities supporting distributed software systems. Component-Based Software Engineering (CBSE) [Heineman, 01] is becoming an accepted engineering discipline for promoting software reuse throughout the software engineering life cycle. Beyond software reuse, CBSE also offers a promising way to manage the complexity and evolution of the development process through a unique means of information encapsulation and separation of concerns at different abstraction levels.

With the advancement of internet technology, component-based software development has unleashed its impact into the distributed environment, while exhibiting such new features as follows:

a. The scope of component selection and reuse is extended. Consequently, component composition requires a prerequisite discovery process for identifying a matching component.

b. Distributed components are usually heterogeneous with respect to implementation languages, and host platforms. With different type systems or component models, interoperation between components will not be possible without leveraging proper bridging technology.

c. Because of the unpredictability of network transport, not only functional properties, but also non-functional properties (e.g., Quality of Service [Raje, 02] and economical properties such as pricing of service) are of critical concern to guarantee the proper delivery of services offered by the assembled distributed software systems. QoS includes availability, throughput, and access control, to name a few.

d. The coupling between components is loose. A deployed component in a distributed system is subject to frequent adaptation or replacement with a new version to accommodate ever-changing business requirements externally as well as the computing resource status internally. Those requirements can be either functional or non-functional.

1.2 Web Services as a New Paradigm for Component-Based Software Development

Those new features pose new problems for developing software systems based on distributed components. Recent years have seen the emergence of Web Services (WS) technology [Newcomer, 02] as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA) [Colan, 04], the open standard description language XML and transportation protocol HTTP. Consequently, distributed component composition can be achieved by wrapping heterogeneous components with a WS layer for interoperation. Using WS as a common communication vehicle, component interoperation is greatly simplified compared with such bridging technology as CORBA[1], where different interoperation implementations are needed for each pair of components contingent on their underlying implementation technologies.

1.2.1 Problems with Web Services

1.2.1.1 Problems with WS as an evolutionary distributed component-based software development paradigm

While WS enables the interoperation among heterogeneous distributed components, which drives the reengineering of legacy software system into WS applications, existing work in this direction requires either expensive manual effort or is language-specific. With the heterogeneity of legacy software systems in languages and platforms, a language-neutral, automatable process is needed to reengineering legacy software systems to WS applications.

[1] CORBA® – Common Object Request Broker Architecture – http://www.omg.org/corba

1.2.1.2 Problems with WS as a Service-Oriented Computing (SOC) paradigm

In addition to offering an interoperability infrastructure for distributed components, WS also incorporates service discovery infrastructure in accordance with SOA. With problem (a) and (b) being embraced, current WS technology is yet to address the concerns as set forth in (c) and (d). Specifically,

1). in mission critical scenarios such as finance or military, there is a need for guarantee of service availability continuously, rather than shutting down the system for services adaptation;

2). in distributed environments, service consumption experiences are dynamic and desirable to be seamless, thus the customizability of service dynamically is of vital importance in a service-oriented environment.

As such, static component composition is not adequate, and both functional and non-functional property adaptation need to be applied in a dynamic fashion. Along this line, this dissertation presents a dynamic component composition paradigm in WS environment for adapting WS functionally and non-functionally while maintaining the availability of WS.

1.3 Research Objective

To fully unleash the power of WS for reusing legacy software components in an internet scale, this dissertation addresses the problems as described in the preceding section. Specifically, the research objectives are twofold: model-driven reengineering WS and dynamic WS composition. The contribution can be summarized as follows:

1). A model-driven approach to reengineering WS in a systematic, automatable and language neutral manner for WS [Cao-d,05]. A meta-modeling approach is initiated in [Cao, 04] based on marshaling and unmarshaling models using Entity-Relationship (ER) models [Cao-b, 05]. Based on the WS meta-model, a WS domain specific modeling environment can be created for synthesizing WS application code, such as Web Services Description Language (WSDL) ([Cao-c, 03]. [Cao-e, 05]). To our best knowledge, there is no peer work that addresses either systematic meta-model construction, or sufficient model-based WS code generation, while our work represents a comprehensive solution to both issues.

2). A non-invasive, cross-language, adaptable approach to dynamic WS composition [Cao-e, 05]. A dynamic WS composition framework based on .NET[2] is created, for which WS applications are captured at Common Language Runtime (CLR) [Gough, 02], and Common Intermediate Language (CIL) [Gough, 02] is manipulated at Just-In-Time (JIT) compilation time for adapting WS at runtime. Consequently, glue-wrapper code can be instrumented at runtime to achieve dynamic WS composition. Moreover, the run-time composition strategy can be adapted to accommodate either external business rules or internal requirements on computational resources.

1.4 Outline

Chapter 2 provides background information, including a survey of component-based software development approaches. Also included are those technologies that are at the core of the research approaches that are presented in this dissertation, such as Model-

[2] .NET - Microsoft .NET framework - http://www.microsoft.com/net

Driven Architecture (MDA[3]), Model-Integrated Computing (MIC) [Lédeczi, 01], Aspect-Oriented Programming (AOP) [Kiczales, 97], Generative Programming (GP) [Czarnecki, 00], and SOC.

Chapter 3 details the model-driven reengineering of legacy software systems to WS applications. That chapter starts off by elaborating on the motivation for the need a model-driven approach for reengineering legacy software system, then explores the process of eliciting meta-models based on marshaling and unmarshaling models using ER models. Different types of marshaling are identified; model marshaling and unmarshaling rules are made explicit. Based on the WS meta-model, a WS modeling environment is described based on the Generic Modeling Environment (GME) tool [ISSS, 01] for WS code synthesis. That chapter also illustrates the WS code synthesis process using the Builder Object Network (BON) API [ISSS, 01] to illustrate why the MIC based approach for code generation is more flexible than the UML profiler [Booch, 99] based static mapping approach.

Chapter 4 introduces the dynamic WS composition. A dynamic WS composition infrastructure is presented based on .NET CLR, which offers two types of dynamic composition paradigms: assertive and autonomous. WS composition is specified following the syntax of AspectJ [Kiczales, 01] for separating composition specification from WS to be composed, as well as for providing a modularized specification for composing WS handling cross-cutting concerns.

Chapter 5 describes some ideas that can extend the work presented in this dissertation. Chapter 6 presents the concluding remarks.

[3] MDA - Model-Driven Architecture - http://www.omg.org/mda

CHAPTER 2
BACKGROUND

This chapter begins with the definition of software components in Section 2.1, then provides a survey of both component models in Section 2.2 and development techniques for component-based software systems in Section 2.3. The contributions of this dissertation which are presented in the following two chapters are based on the synthesis of this background knowledge and ideas. Particularly, Section 2.3 includes the description of the UniFrame[4] project, which is the research project that the author has been associated with during the past 3 years, and contributes to the motivation of the work presented in this dissertation.

This chapter also includes the description of prior work on automatic Feature-Oriented Domain Analysis (FODA) in Section 2.4 and Aspect-Oriented Generative Domain Modeling (AOGDM) in Section 2.5, which not only represent two important elements of component-based software development themselves, but also further complements the description of UniFrame project. Particularly, that work also showcases the synthesis of the techniques presented in Section 2.2 and 2.3.

2.1 Software Components

The definition of what constitutes a software component has been addressed widely in the literature. Rather than proposing a new definition, we adhere to that given

[4] http://www.cs.iupui.edu/UniFrame

by Szyperski [Szyperski, 02], which characterizes the essential properties of a software component as: 1) a unit of independent deployment, 2) a unit of third-party composition, and 3) has no (externally) observable state. Specifically, this definition leads to the following requirements for CBSE development:

- as a deployment unit, software infrastructures are needed for running a component, such as CORBA, J2EE[5], .NET. These technologies are briefly described in the next section.
- a component needs to specify contractually its interface and context dependency explicitly. These contracts state what functionality the component provides, and also what the component requires from the environment and other components. The component specification can be UML based [Cheesman, 01], or formal methods based [Leavens, 01].
- as a component has no externally observable state, there is no difference between any two copies of a component. This contrasts with an object, which has its own observable state encapsulated together with its behavior. Note that a component is quite often built upon a collection of objects, but it is not necessary that every component is composed of objects: a component can be represented in any programming language style (e.g., an imperative programming language, a logic programming language, or a hybrid language).

Distributed software components exhibit yet one more characteristic: the heterogeneity of environment and language. Moreover, in addition to functional properties, a software system composed of distributed software components may embrace a rich set of non-functional properties [Raje, 02] (e.g., throughput, availability, and end-to-end delay).

[5] J2EE - Java 2 Enterprise Edition - http://java.sun.com/j2ee

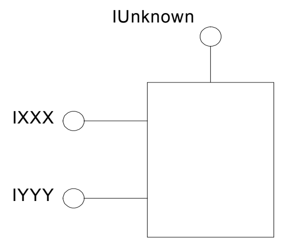
2.2 A Survey of Component Technology Models

This section provides a survey of several popular component models that are analyzed based on the essential characteristics described in the previous section. They are presented in the chronological order of their appearance in the market.

2.2.1 Microsoft COM

COM[6] is the binary standard set by Microsoft for all software components on the Windows platform. Every COM component can implement any number of interfaces, for which an interface called *IUnknown* is mandatory, as illustrated in Figure 2.1-a. All the interfaces for a COM component are specified using Microsoft Interface Definition Language (MS IDL). Each interface distinguishes itself by including a Universally Unique Identifier (UUID) in the MS IDL interface definition. As each COM component must have an IUnknown interface, the UUID for the IUnknown interface can be used to identify the entire COM component. Figure 2.1-b is a sample MS IDL for the IUnknown interface. In an IUnknown interface, the *QueryInterface* is used to identify if an interface is supported or not. If supported, the corresponding reference to the interface is returned. *AddRef* and *Release* in the IUnknown interface are used for maintaining a reference count to the COM component. Once the reference count equals 0, the COM component will perform self-destruction to release the memory space it occupies, and will release all the references it holds to other COM components.

[6] COM - Component Object Model - http://www.microsoft.com/com



```
[UUID(00000001-0002-0003-0004-000000000056)]
Interface IUnknown{
  HRESULT QueryInterface ([in] const IID iid,
    [out, iid_is(iid)]IUnknown iid);
  unsigned long AddRef();
  unsigned long Release();
}
```

(a) (b)

Figure 2.1: COM component

The Distributed COM (DCOM) component model extends COM with distribution based on the Remote Process Call (RPC) mechanism. Microsoft Transaction Server (MTS) further extends DCOM with a container adding transaction and other services, which constitutes the COM+ component model.

2.2.2 Sun Enterprise JavaBeans (EJB)

EJB[7] is the server-side component model for developing enterprise business applications in Java. It is tailored to Java-based applications, which reduces some complex features that are inherent in CORBA (to be described later) for multi-language, cross platform interoperability. An EJB is contained in an EJB Container running on a J2EE Server. The container provides added services to EJB, such as transactions and security. In order for the remote client to be able to access the EJB component executed in the container, distributed objects are used in EJB providing an object-oriented composition model. As such, every EJB (except the Message-Driven Bean, which is explained in the

[7] EJB - Enterprise Java Beans - http://java.sun.com/products/ejb

following part) consists of an EJBHome, EJBObject and EJB Class as are illustrated in Figure 2.2: the EJB class is the core part of EJB representing the business logic; EJB-Home and EJB Object are both distributed objects, the former acting as a factory for the later, with the EJB Object representing the access point for the client to call into the methods of the EJB Class. Also included as part of the EJB component is an XML[8] deployment descriptor file specifying the deployment attributes for the EJB component. An EJB contains three types of beans:

- Entity Bean **-** Entity beans can be shared by multiple clients concurrently. An entity bean is represented as an object, which maps to a persistent data source (e.g., a database). The synchronization between the entity bean and the persistent data source is managed by either the bean itself or the container.
- Session Bean **-** A session bean is initiated by a single client to handle a specific request. If multiple interactions are involved between the client and the Session Bean, then usually a *stateful session bean* can be used to maintain the states throughout the interactions. Otherwise a *stateless session bean* can be used to handle each client request. The transaction for the session bean can be managed either by the bean itself or the container.
- Message-Driven Bean (MDB) **-** A MDB was introduced in EJB 2.0, which is based on JMS (Java Message Service) for data-driven component composition, as opposed to object-oriented component composition used in Entity Beans and Session Beans. An MDB does not require an EJBHome or EJBObject interface. A

[8] XML - eXtended Markup Language - http://www.w3.org/XML

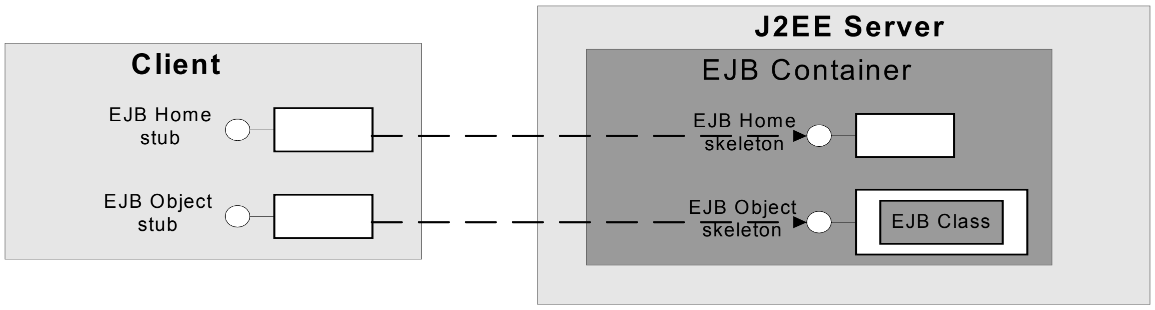


Figure 2.2: EJB component for Entity Bean and Session Bean

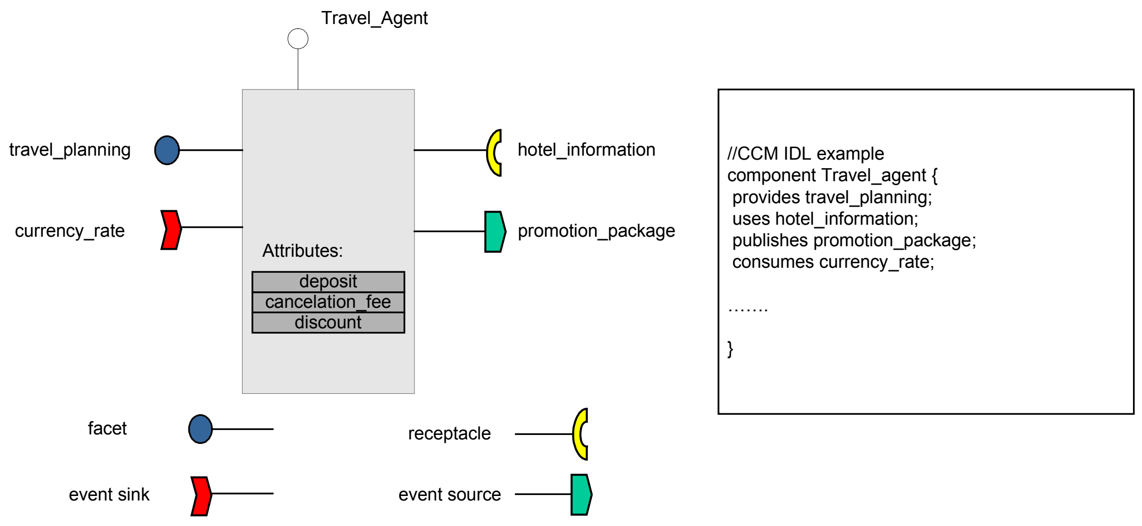
MDB defines message handlers and needs to be registered to a message queue in order to be used for handling messages sent by client applications.

2.2.3 OMG CORBA Component Model (CCM)

CORBA is the initiative of the OMG[9] for enabling interconnections among distributed software components across heterogeneous platforms. CCM[10] was introduced with CORBA 3.0. In contrast to the prior CORBA object model, CCM is designed for loose coupling between CORBA objects, facilitating component reuse, deployment, configuration, extension and management of CORBA services. Figure 2.3 shows an example of a travel agent component represented in CCM. The essential elements within a CCM component are:

- *facets*, which define provided interfaces that the component exposes to clients.
- *receptacles*, which define the required interfaces for the component to function appropriately.

[9] OMG – Object Management Group - http://www.omg.org
[10] CCM – CORBA Component Model - http://www.omg.org/technology/documents/formal/components.htm



```
//CCM IDL example
component Travel_agent {
 provides travel_planning;
 uses hotel_information;
 publishes promotion_package;
 consumes currency_rate;

…….

}
```

Figure 2.3: CCM component

- *event sources*, which publishes the events to clients.
- *event sinks*, which consumes the events published from clients.
- *attributes*, which are used mainly for component configuration.

The facets, receptacles, event sources and event sinks are *ports* for a CCM component model that offer a connection-oriented composition model. The difference between the facet-receptacle connection and event source-sink connection is that the former is connected through an object reference, but the latter is connected through an event channel. Also illustrated in Figure 2.3 is the corresponding IDL for the travel agent component. Similar to EJB, CCM components can be categorized into four types: service, session, entity and process components. Service components are stateless corresponding to a stateless session bean of EJB; session components maintain states for the duration of each session corresponding to a stateful session bean of EJB. Both entity and process components have persistent state, but the former has a lifecycle beyond a specific process and the latter has a lifecycle that is per-process based. A CCM component also runs within a CCM container, which provides added services (e.g., transactions, security, and persistence) to each CCM component.

2.2.4 Microsoft .NET component model

In the Microsoft .NET framework, an assembly is a component that runs on Microsoft CLR. Each .NET language (e.g., C#, VB.NET, C++.NET) can be compiled into assembly files in the form of intermediate code, which are further compiled just-in-time into native code that can be executed in the CLR. Although an assembly component relies on type information for specifying component interoperation (using contracts as in COM), the interoperability for an assembly is at the logical, intermediate code level rather than strictly at the physical, binary level. This makes assembly components easier to use and integrate when compared to COM components. Specifically, the contract specification for an assembly component is represented with machine readable, fully formatted *metadata* embedded together with the MS CIL code inside an assembly. CIL is based on Common Type System (CTS) [Gough, 02].The metadata can be readable and writable by CLR. It can also be extendable by user applications through custom-attributes. Also included in the metadata is the component dependency information described with a *manifest* of the names of adjunct *modul*es[11]*/assemblies*, each providing ex-

[11] A .NET application can be compiled either as a module or an assembly. But a module has to be affiliated with an assembly in order to be deployed. Thus only an assembly can be treated as a complete component. An assembly can be composed of multiple modules together with references to multiple dependent assemblies.

tra type definitions and code. Figure 2.4 illustrates a manifest for an assembly component "HumanResource". Enclosed in each assembly block in Figure 2.4 are a public key token and version information, which are part of the strong name schema for an assembly component naming. The strong name acts as a UUID in a COM component for resolving the assembly component reference when loaded by CLR. Note the dependency specification is missing in the COM component specification. The CLR, as the assembly component execution environment, further makes use of the ubiquitous metadata for *managed execution*, providing appropriate memory management and code verifiability for ensuring system security.

```
.assembly extern mscorlib
{ .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .hash = (E6 8E F4 00 2B 3C 3C 88 D6 32 F2 72 A3 22 FA C8
          A7 7B 24 07 )
  .ver 1:0:5000:0 }
.assembly extern Payroll
{ .publickeytoken = (CA 87 F9 84 99 97 A5 37 )
  .ver 0:0:0:0}
.assembly extern System
{ .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:5000:0}
.assembly HumanResource
{ .custom instance void
[mscorlib]System.Reflection.AssemblyKeyFileAttribute::.ctor(string)=(/…/ )
   // ...keyPair.snk..
  .publickey = (/…./ ) // ignored for saving space
  .hash algorithm 0x00008004
  .ver 0:0:0:0}
.file Travel.netmodule
    .hash = (FE D5 17 E3 9E 25 55 1F 56 F0 1F AF 97 5E 2C 62 34 F9 8D 10)
.class extern public Expense
{ .file Travel.netmodule
  .class 0x02000002}
.module HumanResource
```

Figure 2.4: .NET component specification - a manifest of an assembly component of human resource management: the italicized part represents the specification for dependent modules/assemblies; the bold-font represents the main module and the metadata for the assembly.

2.3 A Survey of Component Development and Composition Techniques

This section describes several component development and composition techniques based on state-of-the-art software engineering ideas. These paradigms can be applied across different component models rather than being restricted to a specific component model.

2.3.1 Model Driven Development (MDD)

MDD uses higher-abstraction models for developing lower-abstraction software applications. Two representative MDD paradigms are MDA by OMG and MIC by Vanderbilt University.

2.3.1.1 Model Driven Architecture (MDA)

MDA is an initiative from OMG for capturing the essence of a software system in a manner that is independent of the underlying implementation platform. MDA can assist in reengineering legacy software systems and Commercial-Off-The-Shelf (COTS) software into Platform Independent Models (PIMs). A PIM can be mapped to software components on Platform Specific Models (PSMs), such as CORBA, J2EE or .NET. In this way, legacy systems and COTS components can be reintegrated into new platforms efficiently and cost-effectively [Frankel , 03]. Figure 2.5 illustrates the whole process in MDA. The vision of MDA also includes standards that enable generative construction of interoperating bridges between different technologies leveraging application and platform knowledge. One of the MDA technologies is an Interworking Architecture[12], which provides a bridge that allows COM and CORBA objects to interoperate from model-driven

[12] http://www.omg.org/cgi-bin/doc?formal/02-06-21

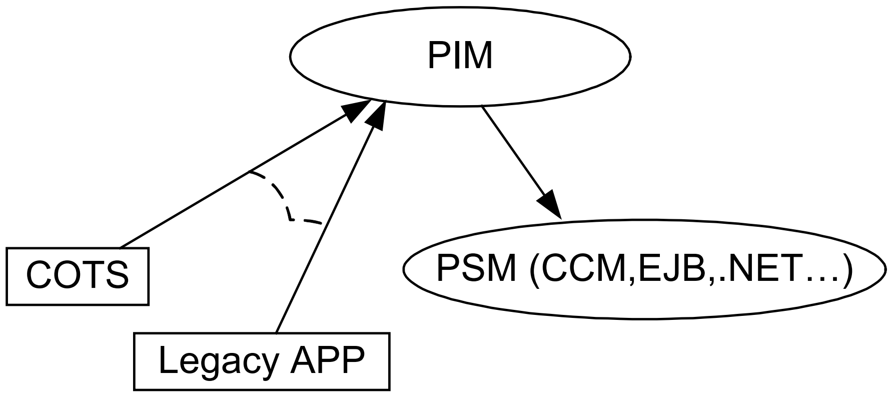


Figure 2.5: Model-Driven Architecture for reengineering legacy software to component models

specifications.

2.3.1.2 Model-Integrated Computing (MIC)

MIC is essentially a development paradigm that offers a means for creating a modeling language (*meta-model*), its associated modeling language interpreter (*generator*). Then any domain-specific model built based on the modeling language can be interpreted by traversing the model tree. The result of the interpretation process is the code synthesized from the model. MIC has been widely used in middleware ([Edwards, 04], [Gokhale, 04]) and embedded systems ([Karsai, 03]; [Lédeczi, 03]).

To ease the understanding of MIC, Table 2.1 provides an analog between MIC and conventional programming language elements. Figure 2.6 provides an example of a meta-model of a Finite State Machine (FSM) and the corresponding model based on it.

Table 2.1: Comparison between MIC and programming language

MIC	*Programming Language*
meta-model	grammar
generator	compiler/interpreter
domain-specific model	application developed using the corresponding language
code synthesized in any chosen language	intermediate code or native code

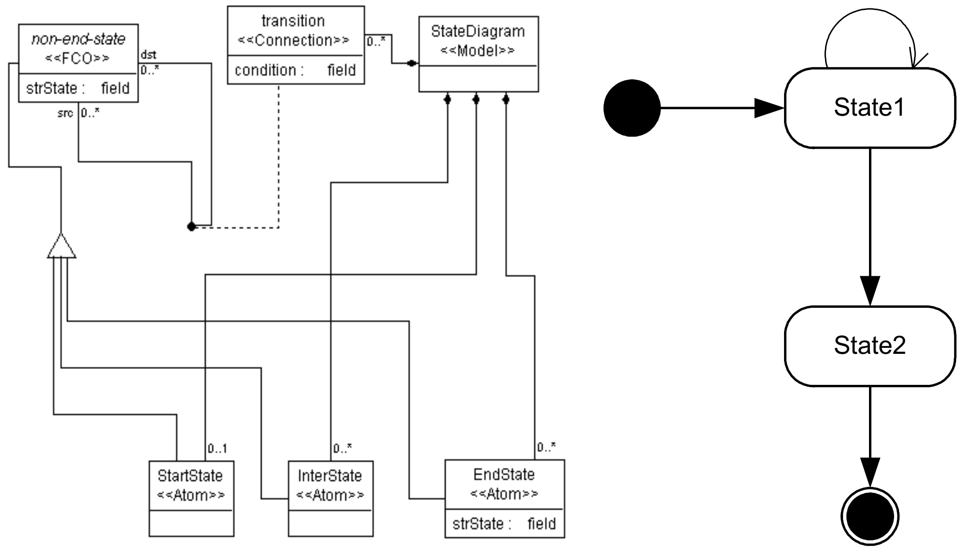


Figure 2.6: A simple example of meta-model and model – the left one is a meta-model Finite State Machine (FSM); the right one is a model of FSM

Furthermore, MIC includes the Generic Modeling Environment (GME) [ISIS, 01] for creation of domain-specific models, a Model Database for model storage, and a Model Interpretation technology for building model interpreters, which can be used to synthesize implementation code from models. In GME, the meta-models use Unified Modeling Language (UML) class diagrams [Booch, 99] to model the system information.

2.3.2 Generative Programming (GP)

GP, as introduced by Czarnecki [Czarnecki, 00], is a software engineering paradigm which uses automation to generate a family of elementary implementation components; a concrete software system can then be produced automatically based on configuration over the elementary implementation components. Specifically, GP contains two-levels of abstraction: at the higher level is the *problem space* that includes the family members and the requirements specifying a software system from the family members; at the lower level is the *solution space*, which is composed of elementary implementation components and their configuration knowledge (e.g., minimum redundancy, mutual exclusion). The production of a software system is firstly ordered in the problem space, which in turn maps to the solution space for implementing a software system product. The problem space and solution space constitute the Generative Domain Model (GDM).

2.3.3 Aspect-Oriented Programming

For component assembly, there are compatibility concerns related to interface issues of component connection, as well as concerns that crosscut the modularization boundaries of individual components (e.g., Quality of Service (QoS), distribution, and synchronization). Consequently, there is a need for capturing those concerns in a modular way. The idea of Aspect-Oriented Programming (AOP) [Kiczales, 97] can be applied to CBSE.

AOP provides a means to capture crosscutting aspects in a modular way with new language constructs: an *advice* is used to represent the cross-cutting behavior, and a *join point* (a collection of which is called *pointcut*) is used to specify the location in the base program to apply the advice. Finally, a new type of translator called a *weaver* to compose the aspects into the base components. Aspects have a direct application to CBSE, which is described in the remaining part of this section.

2.3.3.1 Aspectual Components: a language solution

In AOP languages such as AspectJ [Kiczales, 01], join points are represented by referring to the syntactical constructs of the base program source, thus advices are bound to the base program statically and hinders reuse, which is against the vision of reuse that CBSD promotes. In [Lieberherr, 99], the concept of *aspectual component* is defined, for which aspects are decoupled from the base program by being defined as a generic aspectual component, which is instantiated later over a concrete data-model using a *connector* construct. The concept of aspectual component fosters the integration between AOSD (Aspect-Oriented Software Development) and CBSD. Below are some existent work that follow the *aspectual component* paradigm.

- In [Suvée, 03], the JasCo language is introduced, which introduces two contructs: *aspect beans* and *connector*. *aspect beans* describe functionality that crosscut components, for which a *hook* is defined to represent the association between *join point* and *advice*: The former is represented by method parameter and the method parameter is instantiated in the *connector*. By applying Java binary code transformation to existent binary Java Beans code to add traps to every method that a bean implements, the *connector* registry will be queried at run time for *hooks* (which is a language construct representing aspect definition template), and consequently advices defined in hooks are weaved and executed.

- In [Choi, 00], an Aspect-Oriented EJB Server (AES) is developed for which such functionalities as transaction, persistence, security are represented as built-in aspects (corresponding to an *aspectual component*), and the bean container is changed to generalized *metaobjects* possessing full control of the *baseobject* and delegating method calls of the *baseobject* to the related build-in aspects. The responsibility of the *metaobject* pretty much covers the part that is done by the binary code transformation tool as well as the *connector* in the JasCo component model.

2.3.3.2 Aspect-Oriented Component Engineering (AOCE): an engineering solution

In [Grundy, 00], the concept of horizontal slices through vertically-decomposed components is used to characterize crosscutting properties of components. The aspects in AOCE have a broad definition, which include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, data management, component inter-relationship, and configuration characteristics. Each aspect is comprised of a number of properties describing functional and non-functional characteristics. Based on the aspect characterizations, multiple perspectives of a component-based system can be obtained, and reasoning about component interaction in a variety of ways can be achieved. AOCE, as an engineering approach, covers the lifecycle of component engineering, from component requirements and specification, to implementation, deployment, and testing. In contrast to AOP, which highly relies on code weaving, AOCE aims to use aspects to support component provisions.

2.3.3.3 Other related work on aspects in CBSE

In [Duclos, 02], non-functional aspects are separated from components themselves to promote component (and non-functional aspect) reuse. The non-functional aspects are handled by the Aspect Definition Language (ADL) and Aspect Use Language (AUL). The advice specified in ADL will be accomplished by changing the behaviors of the Component Virtual Machine (CVM). In [Göbel, 04], a COMQUAD component model is introduced that enables the specification and runtime support of non-functional aspects, which is woven into the running applications by the component container acting as a contract manager.

2.3.4 Service-Oriented Computing

Web Services (WS) have emerged as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA) [Colan, 04] as illustrated in Figure 2.7.

By using XML as a standard description language and HTTP as a transport protocol, services can be used to wrap legacy software systems to be integrated beyond the enterprise boundary across heterogeneous platforms. To be specific, WS uses the XML based Web Services Description Language (WSDL) for specifying services, SOAP (Simple Object Access Protocol) messages for service invocation, and UDDI (Universal Description, Discovery and Integration) registry for service discovery [Colan, 04].

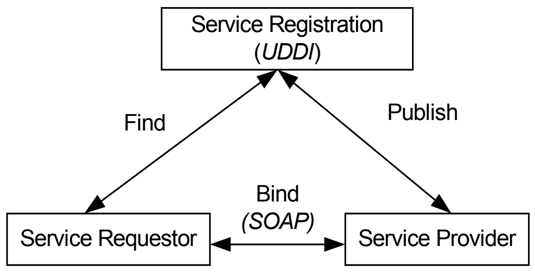


Figure 2.7: Service Oriented Architecture (SOA)

Figure 2.8 provides the WSDL structure in a class diagram. The WS *message*s, which are either *input* or *output* messages, are composed of *port*s, each of which corresponds to a specific data *type*. The *portType* is an abstract WS interface definition, where each contained element (i.e., the *operation*) defines an abstract method signature. The operation uses messages as its parameters. *Binding* represents an instantiation to the abstract *portType* with a concrete protocol and data type. *Service* is a collection of *port*s, denoting a deployment of a binding at a specific network location. The WS orchestration languages, such as the BPEL4WS[13], can be used to encode how different WS work together cooperatively to realize a type of component composition. Note that in contrast to conventional distributed components, WS represents a stateless, loosely coupled computing model. With the increasing popularity of WS, more vendors are either using WS as a presentation layer for back-end components or providing infrastructural support for WS.

The above listed component development and composition paradigms are not applied in isolation, but rather contribute to each other. For example, an AOP approach can be used at the software component design phase for capturing crosscutting concerns at

[13] BPEL4WS - Business Process Execution Language for Web Services - http://www-128.ibm.com/developerworks /library /specification/ws-bpel

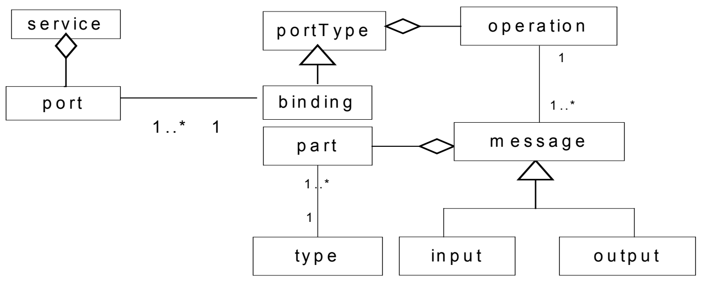


Figure 2.8: Architecture of WS description elements

the PIM level, which later can be weaved together into a PSM. This is similar to the approach adopted in domain-specific aspect-oriented modeling [Gray, 01]. Additionally, AOP can also be used to refine the granularity of the GDM [Cao-a, 05].

2.3.5 Software Factory

With new component models and infrastructure emerging each year, CBSE is becoming more complicated from the viewpoints of design, implementation, and deployment. Nevertheless, the goal of CBSE is not only to promote software reuse, but also to boost the industrialization of software components in a manner similar to the success of hardware components. Toward that end, the concept of a *software factory*[14] has recently been introduced [Greenfield, 04]. A software factory is defined as a "*software product line that configures extensible tools, process, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-*

[14] The term software factory is overloaded; the same term was used by Michael Evans in his 1989 book *The software factory : a fourth generation software engineering environment*. We use the concept as defined in [Greenfield, 04].

based components" [Greenfield, 04]. A complementary vision is also described in Uni-Frame to automate the component assembly with non-functional property constraints based on domain knowledge, which is described in the next section.

2.3.6 UniFrame

UniFrame is a framework to for assembling heterogeneous distributed components with non-functional property guarantee. It uses a Unified Meta-component Model (UMM) [Raje, 00] to encode the meta-information of a component such as functional properties, implementation technologies, and cooperative attributes. In UniFrame, a GDM is also used to capture the domain knowledge and to elicit assembly rules. But the use of a GDM does not include the implementation components: this part is assumed to be ordered in a distributed system environment by different vendors observing the stipulated specifications in the problem space of the GDM; those implementation components are exposed by vendors and are subject to location by a distributed resource discovery service [Siram, 02]. In addition, the GDM in UniFrame is used to capture the assembly rules for the discovered components. Figure 2.9 illustrates the big picture of UniFrame. The annotated number represents the processing order. Starting from domain experts, a GDM will be created (1.1) and will be used together with some domain standards (1.2) as guidelines (2.1, 2.2) for component developers to implement components in solution space. Those implementation components, after being quantified with some QoS parameters (3), will be exposed to a distributed resource discovery service (5). Thereafter, a system integrator will query into the problem space of the GDM for available/deployed component information (6), and then

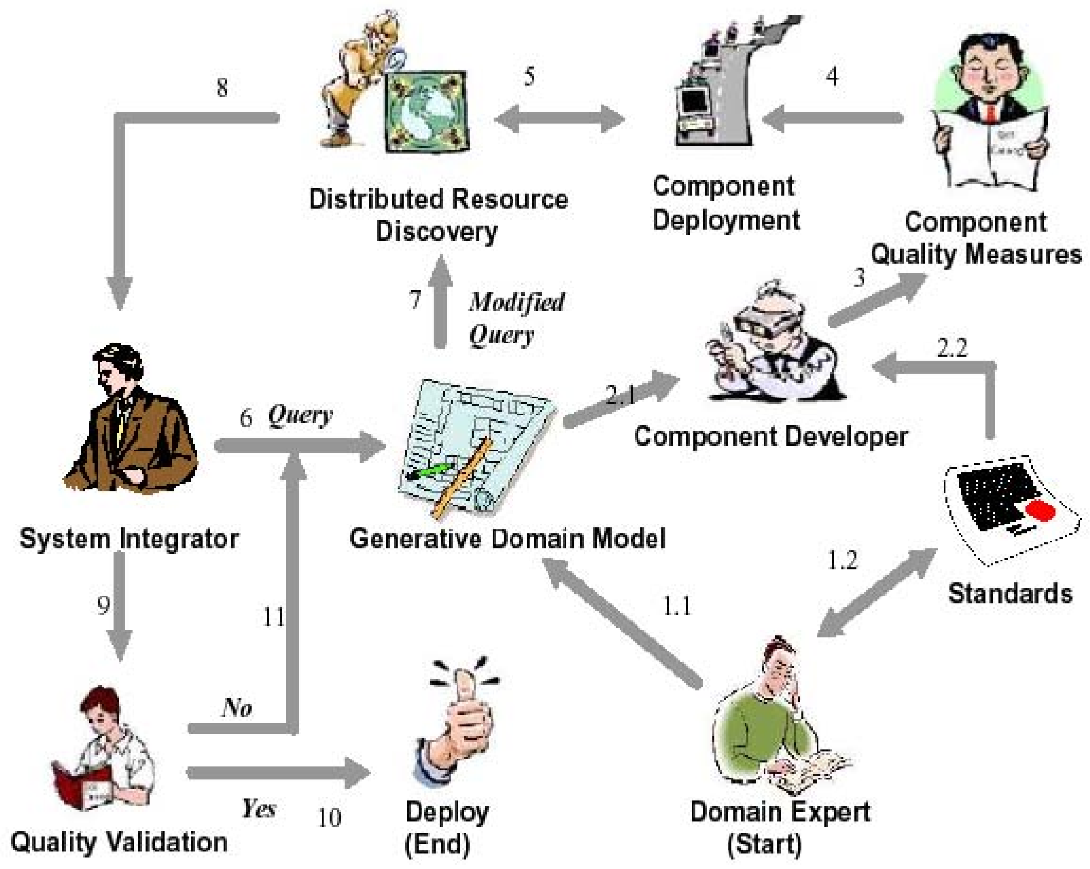


Figure 2.9: Process of Uniframe

request the resource discovery service (7) to fetch the required components (5,8) for assembly. The component assembly is subject to validation (9) based on specified QoS requirements. If it is not validated (11), then the integrator has to initiate the query and integration process iteratively. As it can be seen from above, the GDM stands as a crucial part of UniFrame, and how the GDM is represented so as to facilitate the component assembly is of vital importance.

The following two sections further details the derivation of GDM using automatic FODA (the first account of this part can be seen in [Cao-a, 03]), and the use of AOP to refine the granularity of GDM to support generative multi-stage component assembly (the first account of this part can be seen in [Cao-b, 03]).

2.4 Automatic Feature-Oriented Domain Analysis for a Family of Components

2.4.1 Feature-Oriented Domain Analysis (FODA)

To build a GDM, domain analysis has to be applied to scope a system family and to identify the commonalities, variabilities and dependencies among family members. Consequently, a family of components can be derived based on FODA. A crucial outcome of the domain analysis phase is a feature model, which describes mandatory, alternative or optional features configuration of a stakeholder, as is illustrated in Figure 2.10 by feature diagrams.

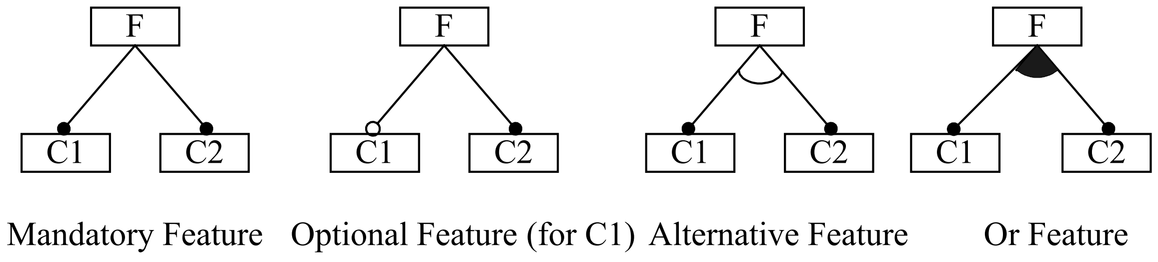


Figure 2.10: Feature diagram representation

The *mandatory* feature is represented by being attached to an edge ending with a filled circle. So the feature F consists of both C1 and C2 in this case, and the feature instances here are {F, C1, C2}. The *optional* feature is represented by being attached to an edge ending with an unfilled circle. So the feature F may or may not contain C1. The *op-*

tional feature instances here are {F, C2} and {F, C1, C2}. The *alternative* feature is represented by connecting edges with an arch. So the feature F consists of exactly one of its child features. The *alternative* feature instances here are {F, C1} and {F, C2}. Note that if C1 is optional while C2 is mandatory, then the *alternative* feature instances here are {F}, {F, C1} and {F, C2}, because the child feature instances derived from the C1 side contain an empty feature. The *Or* feature is represented by connecting edges with a filled arch. The *Or* feature instances here are {F, C1}, {F, C2} and {F, C1, C2}. If there is an optional child feature, then the *Or* representation is actually equivalent to the situation that all the child features are optional, i.e., the *Or* feature instances will be {F}, {F, C1}, {F, C2} and {F, C1, C2}.

2.4.2 The need of automation for Feature-Oriented Domain Analysis

However, the application of feature diagrams is quite limited, due to the fact that current practice is not fully automated: while the size of the set of feature instances may be expanded exponentially (which is to be exemplified in Section 2.4.2.2), a manual approach to FODA will not scale. In order to align with the vision of GP for the highest level of automation, to cope with large scale family system processing, feature modeling should be carried out in an automatic fashion to seamlessly generate reusable assets to be used in application engineering for constructing a family of applications.

The FODA method in its first occurrence in [Kang, 90] uses Prolog in a prototype tool for doing checking over some sets of feature values. However, features have to be stored in the Prolog fact base first, rather than being analyzed directly over the feature diagram, thus the tool is not seamlessly integrated with the visual diagram setting. A similar drawback is also with that described in [Deursen, 02], where the features are described with textual Feature Description Language (FDL), which in turn is executed in a separate environment "ASF+SDF Meta-Environment" [Brand, 01]. For those two approaches, graphically feature models have to be translated into intermediate feature presentation language for backend post-processing, while such model-to-language translation is lacking in these two works. Czarnecki and Eisenecker [Czarnecki, 00] also explore the possible implementation of feature diagrams by mapping into UML, which in turn may be used to generate some implementation codes using such CASE tools as Rational Rose[15] . The mapping process, however, is again a manual process. Also, what Rational Rose can generate are just some skeleton codes, which are far from being complete implementations.

The following section presents an algorithm for generating the set of all feature instances from a feature diagram. Based on MIC principles, a Generic Feature Modeling Environment (GFME) is created for automating FODA; the aforementioned algorithm is incorporated in the model interpreter for generating feature instance sets.

2.4.3 The algorithm to compute feature diagram

2.4.3.1 Normalization of feature diagram

The representations of feature diagrams in Figure 2.10 are building blocks of an actual feature diagram in practice, which usually intermingles the feature diagrams shown in Figure 2.10. An example is given in Figure 2.11. This mixture form can be normalized so that the father-feature in the feature diagram will only be either *XOR* (corresponding

[15] http://www.rational.com

to *alternative*), or *OR*, or *AND* in relationship to child-features. To illustrate, Figure 2.11 can be normalized into Figure 2.12.

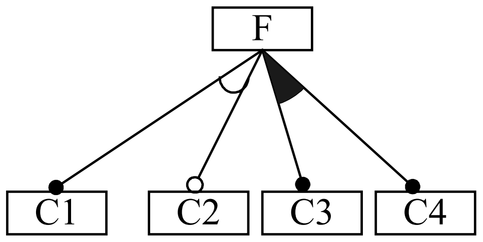


Figure 2.11: Mixture of feature representation

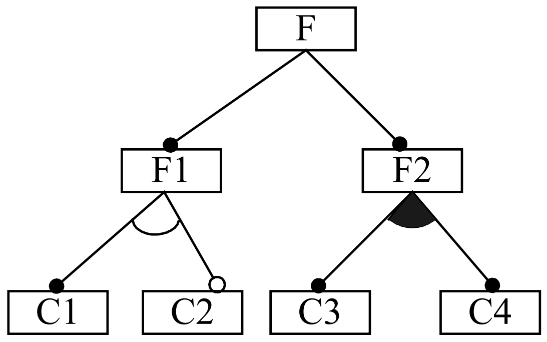


Figure 2.12: Normalized feature representation

This normalization can be performed iteratively over all such "mixture relation" nodes in the feature diagram. Meanwhile, each child-feature may be either *optional* or *mandatory*. Obviously, the normalization process described here is fulfilled by adding hierarchy into the original feature tree without loss of any commonality and variability representations, and normalized feature diagram is easier for representation and process. After normalization is performed, the feature diagram will be in the structure as in Figure 2.13, with

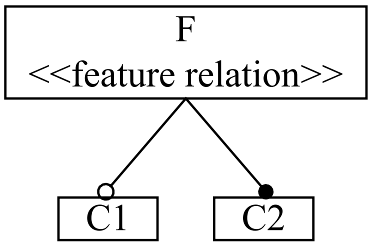


Figure 2.13: Variation of feature diagram - <<feature-relation>> =XOR |OR |AND; C1, C2 may be a sub-diagram

each feature node in the feature diagram being added a meta-attribute *feature relation* to indicate its relationship to child nodes. The proposed algorithm will be applied over such normalized feature diagrams thereafter.

2.4.3.2 Computing normalized feature diagram

Suppose each feature node is represented as the following data structure (note that without loss of generality, the following data structure may not be strictly consistent with a specific C++ programming environment):

```
struct  FeatureNode{
     String featurename;
     enum {XOR, OR, AND} feature-relation;
     /*denotes the father-child  relation */

     ChildConnectionList *edges;
      /*list of connections associated with
       its child-feature nodes */
  }

struct ChildConnectionList {
     bool  isMandatory ;
     /*is a mandatory/optional  feature*/
     FeatureNode * aFeature;
```

```
        /*point to a feature node*/
    }
```

From the data structure above we can see that we can get access to the child-nodes of a feature node by traversing its associated edges. Currently, the result of the algorithm to compute the feature diagram is just the set of all feature instances of a feature diagram. The result will be represented as a list. Each element of the list corresponds to a feature instance. Each feature instance in turn is represented as a list, which consists of the list of pointers to the related feature nodes. The result is represented as follows:

```
typedef List<FeatureInstance *> Result;
typedef List<FeatureNode *> FeatureInstance;
```

Figure 2.14 is the pseudo code for the algorithm. The input parameter to the algorithm is the pointer to the root node of a feature diagram. The output will be all feature instances derived from the feature diagram. Note the variables are in italicized font while the types are in bold font.

Beware that a **Result** is actually a two-dimensional data structure. If **Result** *A* has m **FeatureInstance**s while Result *B* has n **FeatureInstance**s, then the union of *A* and *B* has m+n **FeatureInstance**s while the product of *A* and *B* has m×n **FeatureInstance**s. To exemplify the above algorithm, we use ε to represent an empty **Result**, × for product, ∪ for the union operation in Figures 2.15-2.17, which correspond to three types of cases for computing the set of feature instances. Also from Figure 2.17 we can easily see the size of the feature set may grow exponentially (as to the extreme case where all *feature-relation*s are *OR* , the size will be 2^n, where n is the amount of leaf nodes).

```
Result * processFeatureDiagram (
   FeatureNode *node-root)
 {
 create a temp1:FeatureInstance  with
 only node-root in it;

 create a temp2: Result  with only one
 FeatureInstance temp1 in it;

 if(node-root has no child nodes)
 then  return temp2;

 else
 if (node-root->feature-relation==AND)
  {
  recursively call  processFeatureDiagram
  over each of the node-root's child-
  nodes, each returning a child result;

  if corresponding child node is
  "Optional",
  add an empty FeatureInstance into the
  corresponding child result;

  calculate the production of all the
  returned child results as temp3:Result;

  return the production of temp2 and
  temp3;
  }

else
 if(node-root->feature-relation==XOR)
  {
  recursively call processFeatureDiagram
  over each of the node-root's child-
  nodes,  each returning a child result;

  calculate the union of those returned
  child results as temp3:Result;

  if there is a child node that is
  "Optional",
  add an empty FeatureInstance into
  temp3;

  return  temp3;
  }
```

to be continued in the right pane

```
else
if(node-root->feature-relation==OR)

  {
   recursively call
   processFeatureDiagram
   over each of the node-root's child-
   nodes, each returning a child
   result;

   for each of the child result
   returned in the above call,
   add an empty FeatureInstance into
   it;

   get the production of all the child
   results as temp3:Result;

   If all child features are
   mandatory, remove the empty
   FeatureInstance  from  temp3;

   return the production of temp2 and
   temp3;
   }
}
```

Figure 2.14: Computing normalized feature diagram

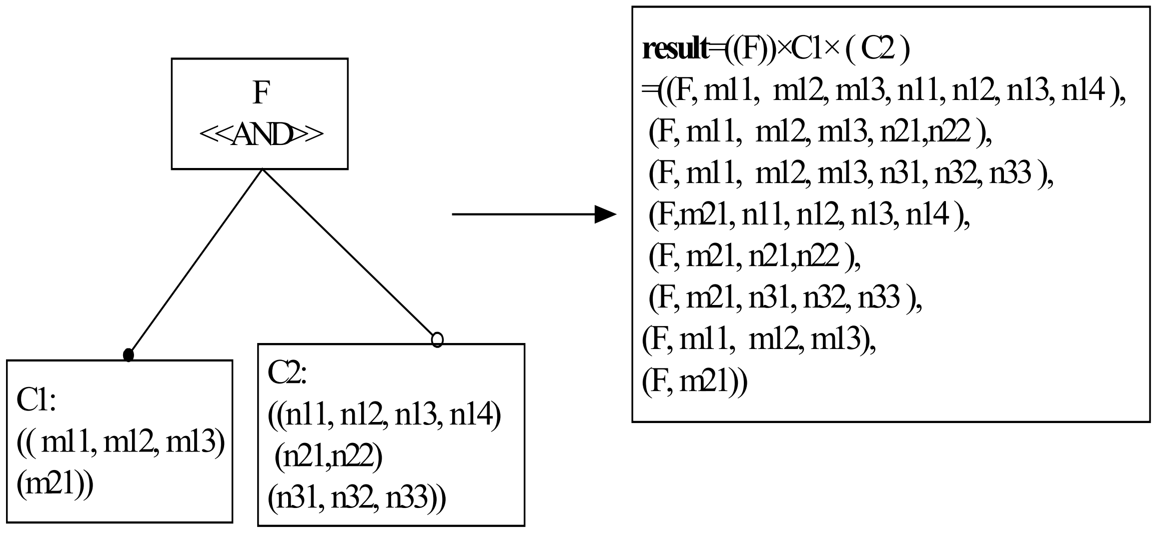

F
<<AND>>
C1:
((m11, m12, m13)
(m21))
C2:
((n11, n12, n13, n14)
(n21,n22)
(n31, n32, n33))
result=((F))×C1× (C2)
=((F, m11, m12, m13, n11, n12, n13, n14),
(F, m11, m12, m13, n21,n22),
(F, m11, m12, m13, n31, n32, n33),
(F,m21, n11, n12, n13, n14),
(F, m21, n21,n22),
(F, m21, n31, n32, n33),
(F, m11, m12, m13),
(F, m21))

Figure 2.15: Computing AND result

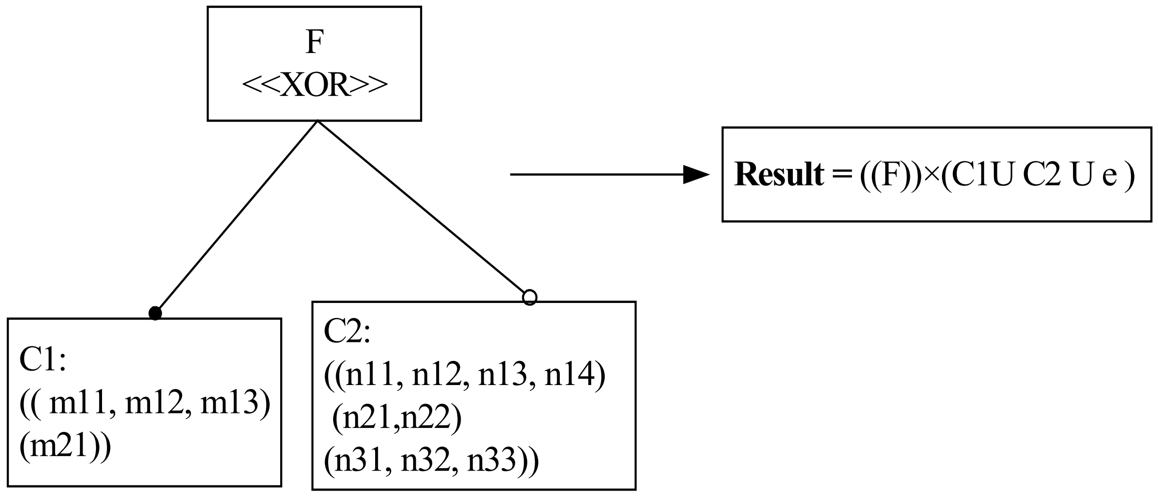


Figure 2.16: Computing XOR result

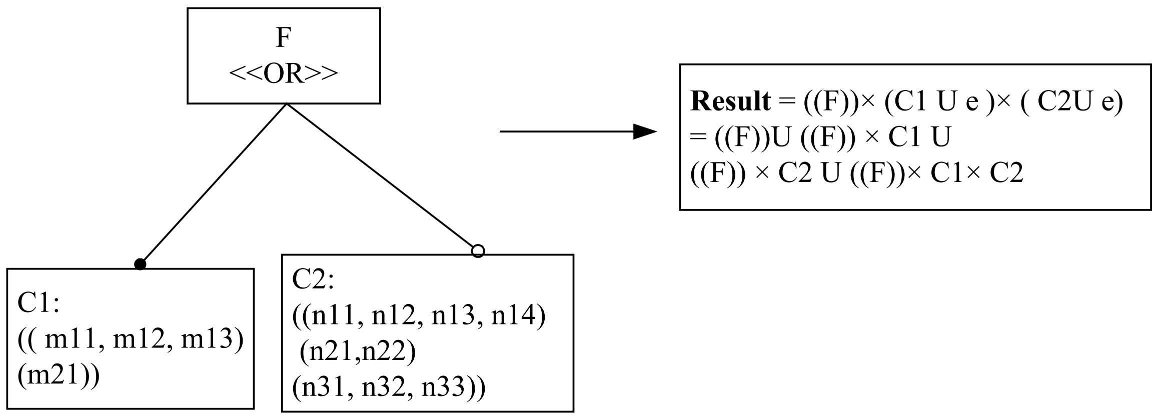


Figure 2.17: Computing OR result

Here we put the non-leaf node (like *F* here) into the feature instances in order to facilitate constraint checking. If one non-leaf feature *F* is supposed to be excluded in the final feature instance, then its child-features should not be included correspondingly, and we can eliminate those feature instances from the final result by identifying which feature instance contains feature *F*, rather than by tracking down all its child-features laboriously.

2.4.4 A Generic Feature Modeling Environment (GFME)

We use GME [ISIS, 01] to build GFME. Figure 2.18 provides the meta-model of the normalized feature model. Each feature atom in the meta-model contains an attribute called the *containment-role*, which represents the containment relationship between this feature and all of its child-features (XOR, OR, AND). Additionally, the connection *Has-Feature* represents the association between the parent-feature and one of its child-feature as optional or mandatory, which is tagged by the boolean variable *isMandatory*. The attribute *containment-role* together with the *HasFeature* connection constitutes the typical feature model representation to describe the commonalities and variabilities of feature model representation to describe the commonalities and variabilities of system configuration. On the other hand, the other types of connection like *mapping*, *interaction* and *mutual-Inc* in the meta-model denote the various kinds of feature interactions [Straeten, 01]. Thereafter, the proposed algorithm will be applied over normalized feature diagrams based on the meta-model as illustrated in Figure 2.18.

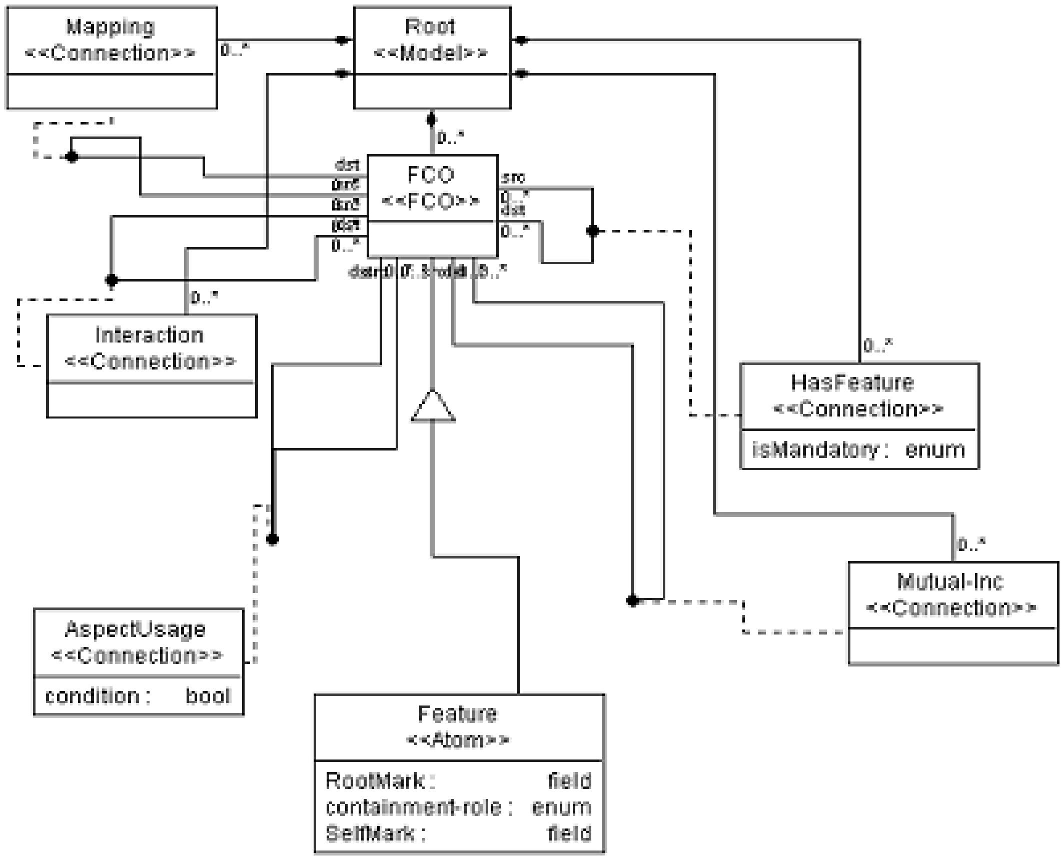
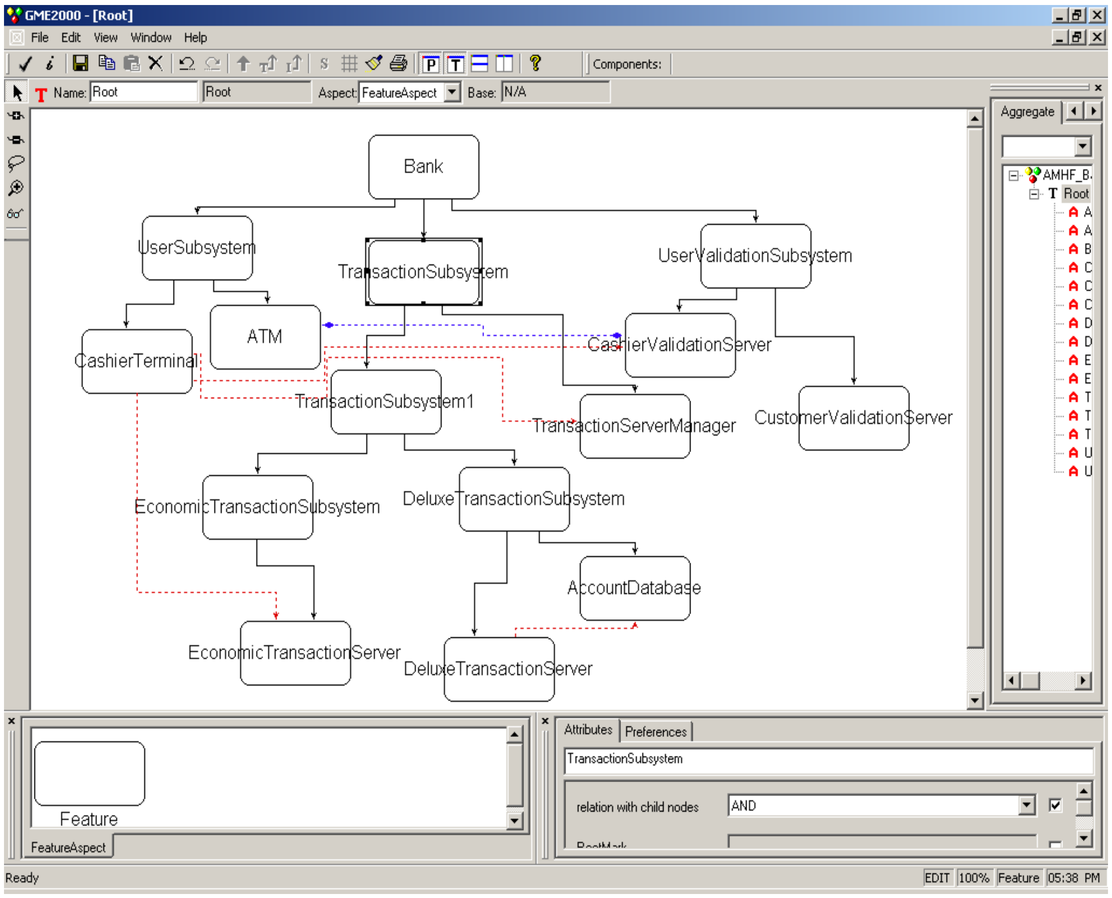


Figure 2.18: Meta-model of normalized feature model

GFME provides the modeling environment for building feature diagrams with the structure as described in Figure 2.18. Figure 2.19 provides the screenshot of the GFME.



Name: Root
Aspect: FeatureAspect
Base: N/A

TransactionSubsystem
relation with child nodes AND

Figure 2.19: Generic Feature Modeling Environment (GFME)

Note at the lower-right corner is the interface to specify such attributes as the relationship with its child-nodes for a node under focus (here *TransactionSubsystem*) in the environment. In the same way, we can specify the attributes for those connections between feature nodes. The dashed lines denote the various kinds of dependencies or constraints to be enforced between feature nodes. Currently we just generate the set of feature instances from the feature diagram satisfying all specified constraints as illustrated in Figure 2.20. With full control of the interpretation process (i.e., writing interpreter code via BON API), we can generate application code from feature diagrams on demand.

```
<?xml version='1.0' encoding="utf-8" ?>
<!--Feature Modeling: generated automatically by feature
metamodel interpreter  @2003/5/23,15:3-->
<architecture_component>
  <system_name>Bank</system_name>
  <case>
    <component>CustomerValidationServer</component>
    <component>CashierValidationServer</component>
    <component>AccountDatabase</component>
    <component>DeluxeTransactionServer</component>
    <component>TransactionServerManager</component>
    <component>ATM</component>
    <component>CashierTerminal</component>
  </case>
  <case>
    <component>CustomerValidationServer</component>
    <component>CashierValidationServer</component>
    <component>TransactionServerManager</component>
    <component>ATM</component>
    <component>CashierTerminal</component>
  </case>
  <case>
    <component>CustomerValidationServer</component>
    <component>CashierValidationServer</component>
    <component>EconomicTransactionServer</
component>
```

to be continued in the right pane

```
    <component>TransactionServerManager</component>
    <component>ATM</component>
    <component>CashierTerminal</component>
  </case>
  <case>
    <component>CustomerValidationServer</component>
    <component>AccountDatabase</component>
    <component>DeluxeTransactionServer</component>
    <component>TransactionServerManager</component>
    <component>CashierTerminal</component>
  </case>
  <case>
    <component>CustomerValidationServer</component>
    <component>TransactionServerManager</component>
    <component>CashierTerminal</component>
  </case>
  <case>
    <component>CustomerValidationServer</component>
    <component>EconomicTransactionServer</component>
    <component>TransactionServerManager</component>
    <component>CashierTerminal</component>
  </case>
</architecture_component>
```

Figure 2.20: Feature instances generated from feature model

2.5 Aspect-Oriented Generative Domain Modeling for Multi-Stage Generative Component Assembly

As is mentioned in Section 2.3.6, the UniFrame uses GDM to capture the domain knowledge and to elicit assembly rules; the GDM includes 1) the solution space which contains the implementation components, and 2) the problem space which is used for external users to query component information and order component assembly.

2.5.1 Specification of components in UniFrame GDM

2.5.1.1 Two-Level Grammar

The components in UniFrame are specified using the formalism of Two-Level Grammar (TLG) [Bryant, 02] as is detailed in the following section. The specification in

TLG provides flexibility in translating TLG specification to other representations, such as other formal specification languages like the Vienna Development Method (VDM) [Lee, 02], or application code [Cao-c, 02]. TLG contains two context-free grammars, one describing type domains and the other describing rules/operations. Note it is not required to have both levels. Below is a simple exemplar TLG specification.

```
class Identifier-1
  Identifier-1,… Identifier-m1 :: DataType1; DataType2;…;
          DataType-n1.
  Function-signature-1,..Function-signature-m2 : function-call-1,
   function-call-2,…,  function-call-n2.
end class Identifier-1.
```

The line containing "`::`" denotes the first-level type domain definition, for which the right hand side of "`::`" provides the type (which is called a *meta-type*) while the left hand side provides the variable name. Note the right hand side may specify multiple types at the same time, which is delimited by "`;`";the left hand side may also have multiple variables separated by ",", which are of the same meta-type as defined on the right hand side. Also note the meta-type may form a hierarchy (*meta-type hierarchy*). For example, *BankOperation* may be the meta-type of *Withdraw* operation, while *Service* may be the meta-type of *BankOperation*. Consequently, *Service* is also regarded as the meta-type of *Withdraw*.

The line containing "`:`" denotes the definition of second-level rule/operation (also called *hyper-rule*) over the first-level type domains. '`;`' can be used in the right hand side of "`:`" to delimit multiple rules which share the same function signature on the left hand side. Note both the first-level and second-level may contain multiple (including zero) sentence as opposed to just one sentence of each in the above example.

2.5.1.2 Component Description Language

Component Description Language (CDL) [16] is used in the problem space of GDM to specify the components, their required and/or provided services in a way to achieve *maximal combination, minimal redundancy, and maximum reuse* (as mentioned in section 2.3.2) as the result of aspect-oriented generative domain modeling. The CDL is also used as a guideline for implementation of components by different vendors. Below is the CDL template.

```
component  <componentname>
  <DomainVariable1>,..<DomainVariable-m> ::
     <DomainType-1>; <DomainType-2>;…; <DomainType-n>.
 [requires <Domain-Specific-Service>: function-call-11,
function-call-
    12,…,      function-call-1n.]
 [provides  <Domain-Specific-Service>: function-call-21,
function-call-
   22,…,  function-call-n.]
end component  <componentname>
```

The first level of CDL provides the type-hierarchy of domain variables. The *re-quires*/*provides* specification constitutes the second level. For the *requires* specification, the right-hand side details the requirement; for the *provides* specification, the right-hand specification further specifies the semantics of the provided services.

2.5.2 Separation of concerns in GDM

Consider the following two component specifications in the GDM problem space.

```
Component BankServer
     provides AccountManagerment:
          applies AccessControl
end Component

Component BankClient
```

[16] Note here CDL refers to both Component Description Language and Component Specification in CDL.

```
    requires AccountManagement:
                  uses RMIServer applying QoSMonitor
end Component
```

In the *BankServer* specification, the provided service *AccountManagement* uses *Access-Control*. But as the business rule is subject to change, the BankServer may lift the AccessControl or enforce other type of controls, either of which will reduce the reusability of original BankServer implementation component. In the *BankClient* specification, the "RMIServer" and "QoSMonitor" that are required for a server-side AccountManagement service represent the glue/wrapping logic, which tangles the BankClient component and also reduces its reusability as glue/wrapping requirements change.

AOP provides a means to capture crosscutting aspects in a modular way with new language constructs, and also provides a *join model* to hook the aspects with the base program. This makes us believe that augmenting the component specification approach with aspect orientation can separate those crosscutting assembly-related aspects of components. The similar idea has been proposed in [Hunleth, 01] to augment CORBA IDL with aspects using AspectIDL, while our approach is based on TLG CDL. Those aspects do not need to be implemented by vendors. The separation will refine the granularity of GDM, and contribute to the *maximal combination, minimal redundancy, and maximum reuse*, which are the desired properties of implementation components [Czarnecki, 00] in the solution space of GDM. Consequently, the component assembly process evolves into an aspect weaving process. Table 2.2 provides the tentative catalog of assembly related concerns.

Table 2.2: Assembly related aspects

Property Type	*Property Attributes*
Functional	Business rule enforcement Specific technology instrumentation Pre/post condition …..
Non-Functional	Profiling QoS Validation QoS Instrumentation …

Figure 2.21 illustrates the aforementioned idea. The arrow ending with a diamond figure represents the *include* relationship in the standard UML notation. Separation of concerns [Parnas, 72] will be introduced into the domain analysis phase, the output of which is GDM. GDM includes the concerns identified at the domain analysis phase (which are also called *early aspects*[17]), and those aspects are collectively stored into a repository called the *aspect library*. This aspect library corresponds to the configuration knowledge in GDM. Upon an *ordering* request over GDM problem space, the CDL in the problem space will be weaved with involved assembly aspects into a glue/wrapper code generation specification, which by referencing the implementation components, will be used to generate final glue/wrapper code.

[17] http://early-aspects.net/

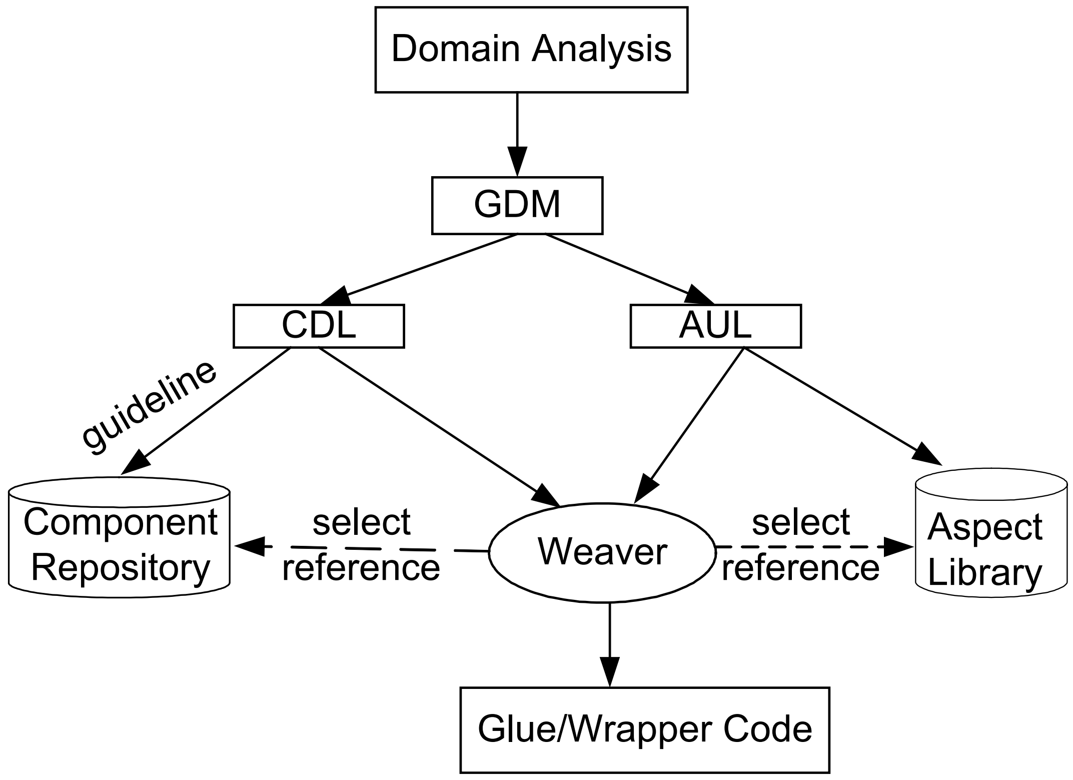


Figure 2.21: Aspect-Oriented Generative Domain Modeling (AOGDM)

2.5.3 Generative multi-stage component assembly

Before we describe the component assembly process in detail in section 2.5.3.3, we provide the related specification definitions.

2.5.3.1 Specification of aspect and the use of aspect

As the aspects as indicated in Figure 2.21 are separately stored as opposed to in such AOP language as AspectJ [Kiczales, 01], where aspects are defined closely bounded to a base program (the *join point* is specified syntactically based on the base program), there needs to be a means to define a *join point model* to hook the aspects to the targeted program, so as to apply the related *advice* provided in aspect defintion to the targeted program.

Aspect Description Language (ADL)[18] is defined as follows:

```
Aspect <aspectname>
     advises: <Meta-type>.
     [before: <advice>.]
     [after: <advice>.]
end Aspect <aspectname>
```

The name enclosed with "<>" represents grammar variable, which will be exemplified below. The "[]" is used to delimit a part that is optional. Those notations apply to the following AUL and CDL. The <Meta-type>, which is defined as in section 3.1, is used to specify the types of domain services that this aspect can be applied to. The *advice* following the directive *before/after* provides the pre/post actions to be performed or pre/post conditions to be enforced before/after the domain services, which can be used as such leverages as temporal dependency specification, tracing/QoS code instrumentation. For example, in [Ubayashi, 02], before/after advices are used to specify rules for model checking. Consequently, the aspect library represents a collection of assembly rules.

Aspect Usage Language (AUL)[19] is defined as follows:

```
     apply <aspectname> on <type> [when <relational expres-
sion>]
```

<aspectname> corresponds to an assembly-related aspect, which already provides a means to specify assembly rules as described in the preceding paragraph. The <type> has to be consistent with the applicable <metatype> in the ADL of <aspectname>. By *consistent* we mean the <metatype> as in the ADL of <aspectname> should reside at the root

[18] Note here ADL refers to both Aspect Definition Language and the Aspect Definition specified in ADL.
[19] AUL refers to both Aspect Usage Language and the Aspect Usage Specification in AUL.

position of some meta-type hierarchy (see section 3.1 for definition), where <type> is also part of it. The *when* directive in AUL further specifies the scenarios using relational expressions, under which this aspect can be applied. It's quite straightforward that AUL can be used in a *product ordering* specification as indicated in section 2.1. Note the definitions of ADL and AUL are inspired by [Duclos, 02], where non-functional aspects are separated from components themselves to increase the component (and non-functional aspect) reuse, and the non-functional aspects are handled with similar language constructs as ADL and AUL described here.

2.5.3.2 Aspectual component as a paradigm for component assembly

The Aspect Library as shown in Figure 2.21 captures the *general* business and technology requirements in terms of assembly-related concerns. However, a component assembly process indicates the *specific* scenario of behaviors in terms of aspect usage (the use of those *general* aspects). Of course the component assembly process cannot be realized simply via a single AUL, as a component captures groups of behaviors. But aspect weaving does offer a means of component assembly in the sense that weaving is also one kind of assembly. So a means is needed to provide another reusable aspect definition model and join point model to the weave aspects and targeted program so as to realize component assembly.

We use the aspectual component model as described in Section 2.3.3.1 for component assembly. However, the original *aspectual component* is in Java, while here it is a language-independent specification in TLG. The *connector specification* classifies server components' related services into some category based on meta-type. The connector specification also includes related operations associated with the meta-type. The meta-type can be regarded as one kind of *join point* in AOP, while the related operations in connector specification provide *advice*. The meta-type, in an aspectual component, is how the client and server component get *hooked up*; the join point model to be used is again type-based as in the preceding section.

We integrate the ideas into a process diagram in Section 2.5.3.3, which is reified by an example in Section 2.5.3.4.

2.5.3.3 Overall picture

Figure 2.22 provides the multi-stage component assembly process. Stage 1 is mainly about the introduction of GDM (from domain analysis), which includes CDL in the problem space and Aspect Library as configuration knowledge. Stage 2 involves the weaving of the aspect specification into component specification for each components involved in the assembly process. Stage 3 illustrates the process of the component assembly specification generation based on the aspectual component model. This stage involves a *connector repository*, where the connector specifications will be registered, and the aspectual component will initiate a *query* into the connector repository to find the matching connector specification based on meta-type consistency, and to apply the associated advice thereafter. The connector specification is translated from the CDLs of the server component (service provider) and the aspectual component specification is translated from a client component (service consumer). Glue/wrapper code will be synthesized in the final stage from the assembly specification, which uses the referencing to the component repository

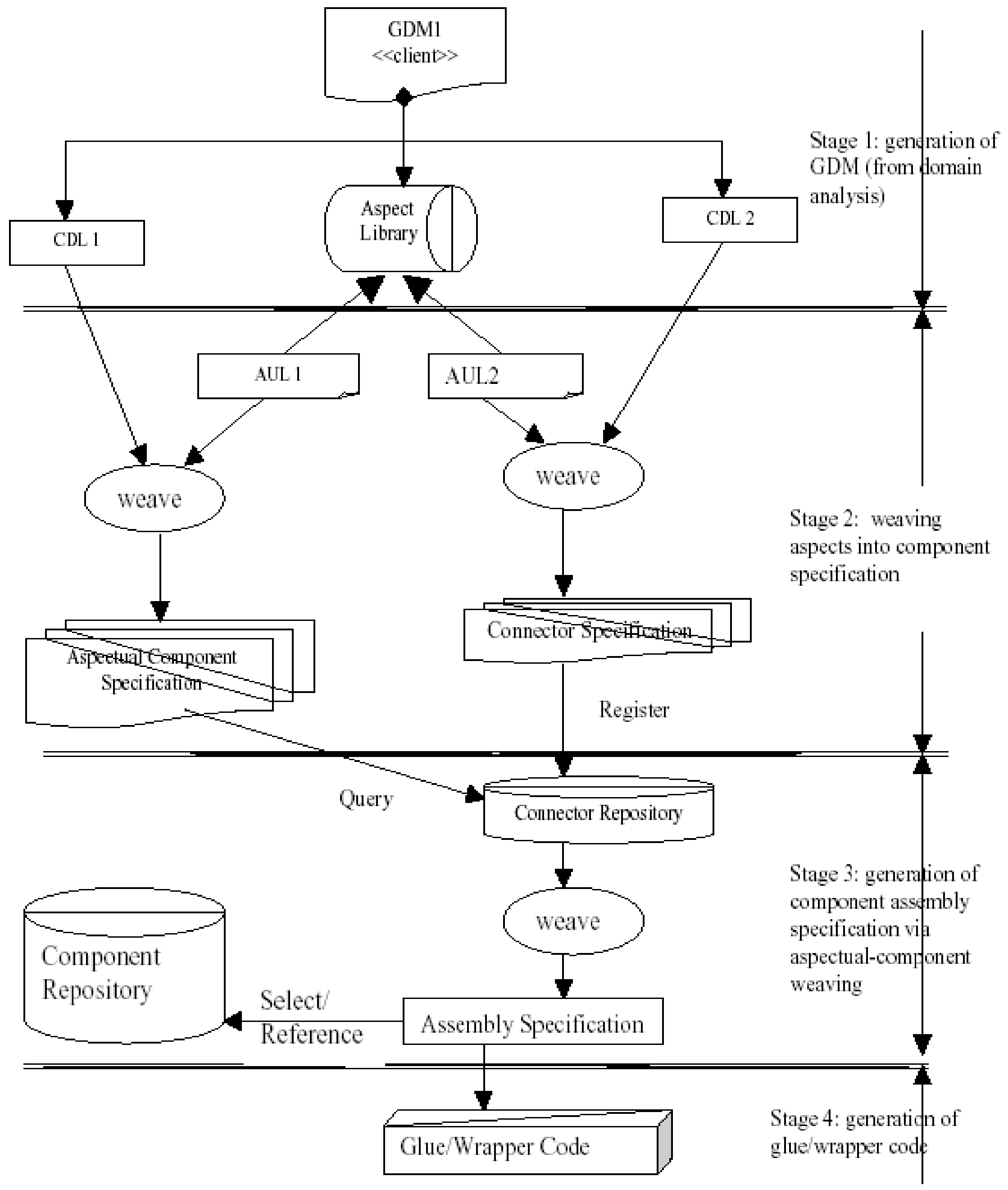

GDM1
<<client>>
Aspect Library
CDL 1
CDL 2
Stage 1: generation of GDM (from domain analysis)
AUL 1
AUL2
weave
weave
Stage 2: weaving aspects into component specification
Aspectual Component Specification
Connector Specification
Register
Query
Connector Repository
Stage 3: generation of component assembly specification via aspectual-component weaving
weave
Component Repository
Select/ Reference
Assembly Specification
Glue/Wrapper Code
Stage 4: generation of glue/wrapper code

Figure 2.22: Multi-stage gluing/wrapping

(which stores the set of component UMM specifications retrieved by the discovery service in UniFrame).

To help clarify the above concepts, a simple generative multi-state component assembly example is provided in Appendix A, demonstrating how the *aspectual component* approach can be adapted to the component assembly process. Note the assembly paradigm described in Appendix A is following a client/server architecture, whereby the client component (service consumer) may be translated into aspectual component specification. In the event the components to be assembled are not following that kind of architecture, the *ordering* specification itself needs to be translated into an aspectual component specification, and then apply the similar assembly process as shown in Figure 2.22.

2.6 Discussion

UniFrame, the motivation project of the presented component assembly approach in Section 2.5, aims at automating the process of integrating heterogeneous components to create distributed systems that conform to quality requirements. GP is the underpinning solution to fulfill this vision. In order to realize the vision of GP for highest level of automation, during domain engineering phase, the creation of the domain model may be applied using MIC in the similar way to GFME presented in Section 2.4. Based on the component assembly approach presented in Section 2.5, Table 2.3 describes the generative programming in UniFrame.

Table 2.3:Generative Programming in UniFrame

Generative Programming	*UniFrame*
Feature modeling	GFME
Components are generated in domain implementation phase	Components are implemented by vendors. Generation only occurs at system level
Configuration Knowledge	Aspect Library
Mapping of problem space to solution space	Resource Discovery Service to search components based on component specification
Domain Specific Language (DSL)	CDL, AUL, ADL
Generator	Aspect Weaver

2.7 Summary

This chapter provides a synopsis of different component models and state-of-the-art component-based software development techniques. In particular, the UniFrame project is introduced, with its GDM and component assembly principles elaborated, which also showcases the synthesis of the component-based software system development approaches described in this chapter. While the next two chapters address the issues of WS technology which represents an evolution of the component software paradigm, the principles and approaches has their roots in UniFrame, and apply to UniFrame reciprocally.

CHAPTER 3

MODEL DRIVEN REENGINEERING LEGACY SOFTWARE SYSTEMS TO WEB SERVICES

This chapter presents a model-driven approach for reengineering legacy software systems to WS applications. This chapter begins with the motivation for a model-driven approach for reengineering legacy software systems, then describes the approach of marshaling and unmarshaling models using ER models for eliciting WS meta-models in an automatable, systematic manner as opposed to an ad-hoc manner. The WS meta-model in turn is used for creating WS modeling environment based on the GME for WS code synthesis.

3.1 Motivation

3.1.1 Definition of legacy software system

With the rapid advancement of software technology, more and more software systems developed with the state-of-the-art technologies of yesterday are becoming legacy software systems of today. Specifically, we define legacy software in a comparative manner, i.e., the software systems are *legacy* if the languages, models or platforms they are developed with can be replaced with new languages, models or platforms of advanced features and improved capabilities. Legacy software systems are heterogeneous in language and platform. With the wrapping of WS, legacy software systems can be reused across heterogeneous distributed platforms.

3.1.2 Approaches for using Web Services as a wrapper

There are several options for reengineering legacy software to WS:

- Manually port original software source code to WS applications. This is an expensive solution. Also WS code, such as WSDL, is verbose, and coding WSDL manually is error prone.
- Language tool based, in which the legacy software package is recompiled to generate WSDL. Many tools such as AXIS[20] and the Microsoft .Net framework provide the function of generating WSDL from implementation code (such as Java and C#) and vice versa. Such tools leverage compiler technology to generate WSDL from other programming languages. The WSDL in turn can be used to generate client side stub code for the client to call the services exposed by legacy software systems [Graham,02]. However, this language tool based solution remains to be language-dependent. With the variety of legacy software systems, a language neutral solution is required in order to sufficiently handle the reengineering of legacy software systems to WS.

As an extension to the preliminary work on a model-driven approach to WS development [Cao, 04], this chapter presents a model-driven approach for reengineering legacy software systems to the WS applications, in which a model plays a central role for migrating legacy software systems to WS implementations. A model is usually represented in UML[21], or any other abundant domain specific visual language (as can be seen in JVLC[22]), which represents the structural and contextual information of a legacy software system in a language neutral style without being tied to implementation specifics. The

[20] http://ws.apache.org/axis/
[21] UML^{TM} - Unified Modeling Language - http://www.omg.org/uml
[22] JVLC - Journal of Visual Languages and Computing-http://www.elsevier. com/locate/jvlc

model-driven reengineering approach is also based on the observation that legacy software systems are usually documented in a visual modeling language; models can also be used as first-class assets in SOA (e.g., model as the basis for service discovery in [Hausmann, 04]).

To apply the model-driven approach for reengineering legacy software systems to WS, a model should play a role beyond the conventional design and documentation capacity, i.e., a role for WS code generation directly to resolve the manual porting problem as described above. Usually UML-based code generation is based on a static mapping from the UML profile [Frankel, 03], which lacks flexibility during the code generation process. As such, we use MIC for building a WS modeling environment and consequently for WS code generation.

3.1.3 Problems for applying MIC to reengineering legacy software to WS

While MIC offers an automatable and language neutral approach for reengineering legacy software to WS, the starting point of MIC - the construction of the meta-model has to be a manual process. Previous work on WS modeling [Cao-c, 03] has revealed that with the increasing complexity of the modeling target, the construction of the meta-model is subject to being ad-hoc and error-prone. With the modeling assets (UML or other domain specific visual modeling language) already abundantly available as part of the legacy software (which we term *legacy model*), it is desirable to derive the meta-model from the legacy model in a systematic, automatable process as opposed to being ad-hoc and error-prone. However, the current meta-modeling languages lack adequate modularity support for large scale meta-model construction, which nevertheless is widely existing in general programming languages. As a result, the construction of a meta-model remains an art rather than a science.

Therefore, the contribution of this chapter is twofold:

1) the elicitation of a meta-model from a legacy model in a systematic, automatable process, and consequently
2) the creation of a domain-specific WS modeling environment for WS code generation, as well as the treatment of WS semantic concerns from a model-driven perspective.

3.2 Marshaling and Unmarshaling Models Using Entity-Relationship (ER) Model

The elicitation of a meta-model from UML or other domain-specific modeling notations can be done on a per source model basis. However, with the constant emergence of new modeling notations, the elicitation approaches will become ad-hoc and not reusable. Moreover, there is a need to converge the diversified modeling assets for modeling tool integration[23]. Therefore, we need to encode the diversified models with a common representation, such that different modeling notations can transfer to and from it, thus modeling assets can be exchanged and used across different modeling tools. We [Cao-b, 05] have referred to these modeling notation transferals as *marshaling* and *unmarshaling*, respectively. The term marshaling comes from the distributed computing scenario where heterogeneous data types are always translated into some common data type over the network so as to be consumed at another end of the distributed environment, where the common data type is unmarshaled again into another environment-specific data type. Comparatively, the concept of marshaling and unmarshaling models refers to transform-

[23] Interview with Keith Short, http://www.theserverside.net/talks/ library.tss#KeithShort

ing a model to an *intermediate common semantic* form, which is reinterpreted in another modeling environment/tool. This intermediate common semantic form is in a similar vein to ACME [Garlan, 00], which is an intermediate form for exchanging software architecture description languages across different software architecture design tools. Moreover, with the heterogeneity of models at different meta-levels (not only model level but also meta-model level) [Frankel, 03], marshaling and unmarshaling of models can be performed at different levels: horizontally, meta-model level and model-level; vertically, meta-model to/from model as is illustrated in Figure 3.1.

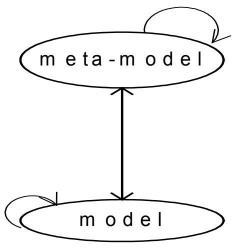


Figure 3.1: Marshaling and unmarshaling models at different levels: the arrow represents marshaling/unmarshaling process

3.2.1 Rationales

Here we use the ER model [Chen, 76] as the intermediate common semantic form for marshaling and unmarshaling models[24]. The rationales are as follows:

- *Sufficiency.* Even though UML is widely adopted in software modeling, which seems to justify the use of UML as a common model for exchanging model assets

[24] Note that the ER model is not intended to replace the existing modeling language such as UML or Petri Nets – those modeling languages have their own advanced features for a specific domain to model. Here the ER model is chosen as an intermediate form only for exchanging models of a close type or serving a close purpose but with variant notations across different modeling tools and environments.

across modeling facilities, UML is not convenient for model serialization, thus not fit for modeling asset exchange, reuse and evolution. In fact, the object diagram [Booch, 99], for which UML is used to capture and store the snapshot of the software system state, is represented virtually in an Entity (object) and Relationship (links) model. Moreover, the UML modeling language has its roots in the ER model, and the latter is already widely used as the foundation for CASE tools in software engineering and repository systems in databases[25].

- *Necessity.* As is illustrated in Figure 3.1, not only models, but also meta-models are in need of marshaling and unmarshaling. Therefore, the intermediate model should be expressive enough to be at the meta-meta model level in the meta-level stack [Frankel, 03]. The meta-meta-model is described by the Meta Object Facility (MOF)[26], which is a set of constructs used to define meta-models. The MOF constructs are the *MOF class*, the *MOF attributes* and the *MOF association*. These constructs correspond to an ER representation (by using an Entity to represent a MOF class), which indicates that the ER representation is semantically equivalent to MOF fundamentally. Therefore, the ER representation is the right vehicle to play the dual roles of marshaling both models and meta-models. Also, other non-UML based languages, even though not as popular, are abundantly present, for which UML is not an omnipotent cure.

[25] http://bit.csc.lsu.edu/~chen/chen.html
[26] Meta-Object Facility - http://www.omg.org/technology/documents/formal/mof.htm

3.2.2 Overview of the approach

The work presented in this chapter aligns with the vertical direction which is further illustrated in Figure 3.2, i.e., marshaling models to the ER model, then unmarshaling the ER model to the GME meta-model. The gray area in Figure 3.2 represents the MIC paradigm. To be specific, in the following section, we will marshal a UML class diagram for Web Services Description Language (WSDL) to the GME meta-model, then create a WS

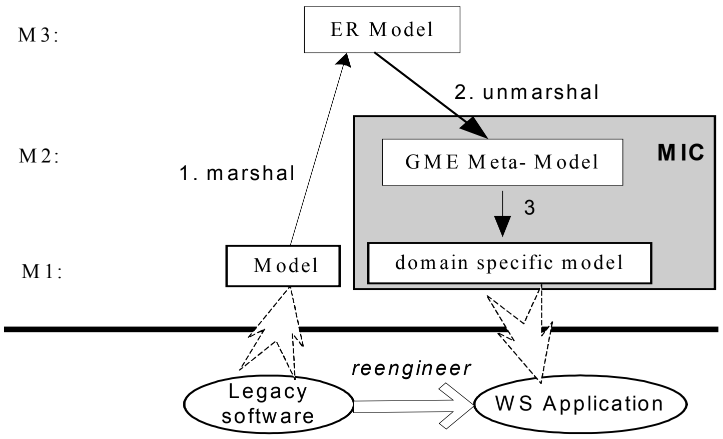


Figure 3.2: Eliciting Meta-models from model via marshaling and unmarshaling models using ER model

modeling environment based on the meta-model for WS code generation. Therefore, legacy software systems can be reengineered to the WS application automatically with a language neutral approach. We also show the generality of this approach: even though the scope is within the vertical direction, the approach can also be applied for horizontal marshaling/unmarshaling using the ER model; even though the source model is the UML object-oriented model, it is not tied to this single kind of source model and can be applied to other domain-specific visual modeling languages as well.

3.3 Reengineering Legacy Software to Web Services

In order to reengineer legacy software to WS, we need to capture 1) the WS technology domain knowledge; 2) the original legacy software business domain knowledge; and 3) original implementation technology information. This categorization of technology domain knowledge and business domain knowledge has been described in [Zhao, 03]. Figure 3.3 describes the legacy banking application information, including its business domain knowledge (the first two paragraphs) and its original technology domain knowledge (the last paragraph). Note as WS is used as wrapper for original technology domain knowledge as well as the original business domain knowledge, rather than replacing the original technology, we treat the original domain knowledge as the part of business domain knowledge in the remaining part of the paper for simplicity.

3.3.1 Marshaling legacy software model to ER model

In order to elicit the banking domain WS meta-model, we need to first merge the WS technology domain information (as illustrated in Figure 2.8) with the business domain information. To that end, we treat the WS technology domain as the *dominant domain* during the merge process, with the business domain knowledge as the *adjunct domain* being appended to the marshaled model from the technology domain model. As such, the marshaling process as illustrated in Figure 3.1 can be decomposed into the marshaling type A for dominant domain and type B for adjunct domain together with a merge

A bank provides the service for users to set up accounts. Account information includes personal data including Name, SSN, phone number, address, and account data including Account Number, PIN, Transaction Record, Balance. There are two types of accounts: checking account and savings account.

For the bank side, it provides such services as: Account Verification, Account Query, Deposit, Withdraw, and Transfer.

The banking service implementation may use such technology as RMI[27], J2EE[28], and CORBA[29]. Also it will enforce some Quality of Service (QoS) requirements such as Availability, Dependability, Capacity.

Figure 3.3: A banking example

step as is illustrated in Figure 3.4.

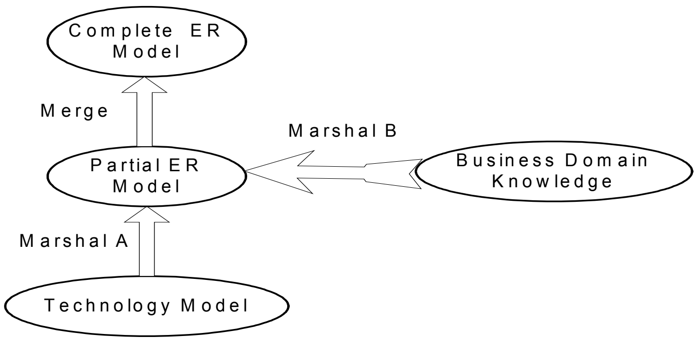


Figure 3.4: Stepwise marshaling

[27] RMI - Remote Method Invocation: http://java.sun.com/products/jdk/rmi/index.jsp
[28] J2EE - Java 2 Enterprise Edition: http://java.sun.com/j2ee/
[29] CORBA® - Common Object Request Broker Architecture: http://www.omg.org/corba/

Table 3.1 illustrates the marshaling rules based on different marshaling types. Note that one of the essential characteristics of a meta-model is that it treats not only the models, but also the inter-relationships among models as first-class entities. Therefore, for marshal type A, the different type of relationships between classes will be mapped to the *Relationship* construct in the ER model, while each class is represented as an *Entity*.

Table 3.1: Marshaling rules

Type	*Rule*
Marshal A	▪ aggregation, association, generalization, and dependency => Relationship ▪ class=> Entity
Marshal B	domain analysis and mapping

Figure 3.5 illustrates the resultant ER model after marshaling the WS class diagram based on this rule. Each diamond represents a type of relationship in the original class diagram. Note we ignore *type* in the ER model of Figure 3.5 because we can put the type directly as the attribute of the part element. However we will not include the attributes to the entities and relationships in the ER representation here, as the focus of this paper is about the model of marshaling and unmarshaling structurally; the attributes will be annotated in the GME meta-model and are shown later. For marshal type B, a domain analysis phase [Czarnecki, 00] is needed to associate the business domain information to the technology domain information. Specifically, the different banking services described in Figure 3.3 can be treated as different types of operations in WSDL, while different banking service implementation technology and QoS requirements can be associated to

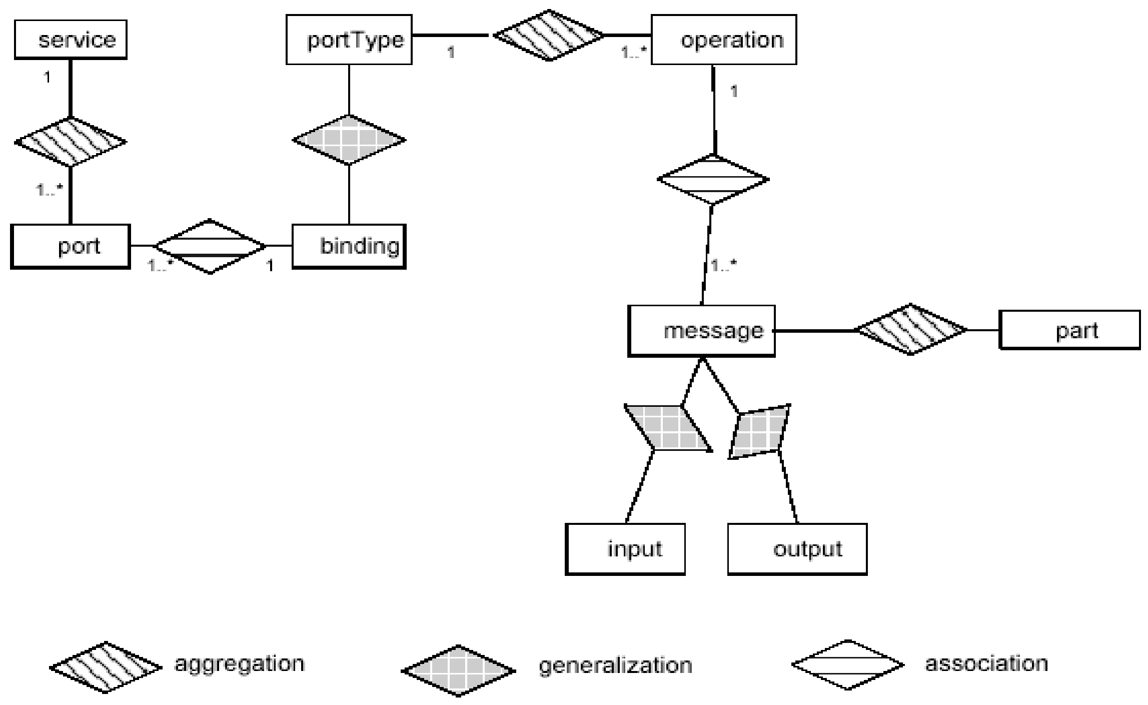


Figure 3.5: Marshaling WSDL model to ER model

bindings in WSDL as a reification of operations. Account information and account type information can be treated as messages in WSDL. Figure 3.6 illustrates in detail the resultant ER model after annotating the business domain knowledge (using either generation relationship or association relationship) to the WSDL ER model illustrated in Figure 3.5. By using the ER model as the intermediate form for marshaling, different types of domain knowledge can be merged incrementally without obfuscating each other, which provides a separation of concerns toward domain-specific model refinement. Also with the non-invasive merge process, the business domain semantics are reified with technology semantics while the business domain semantics are kept unchanged.

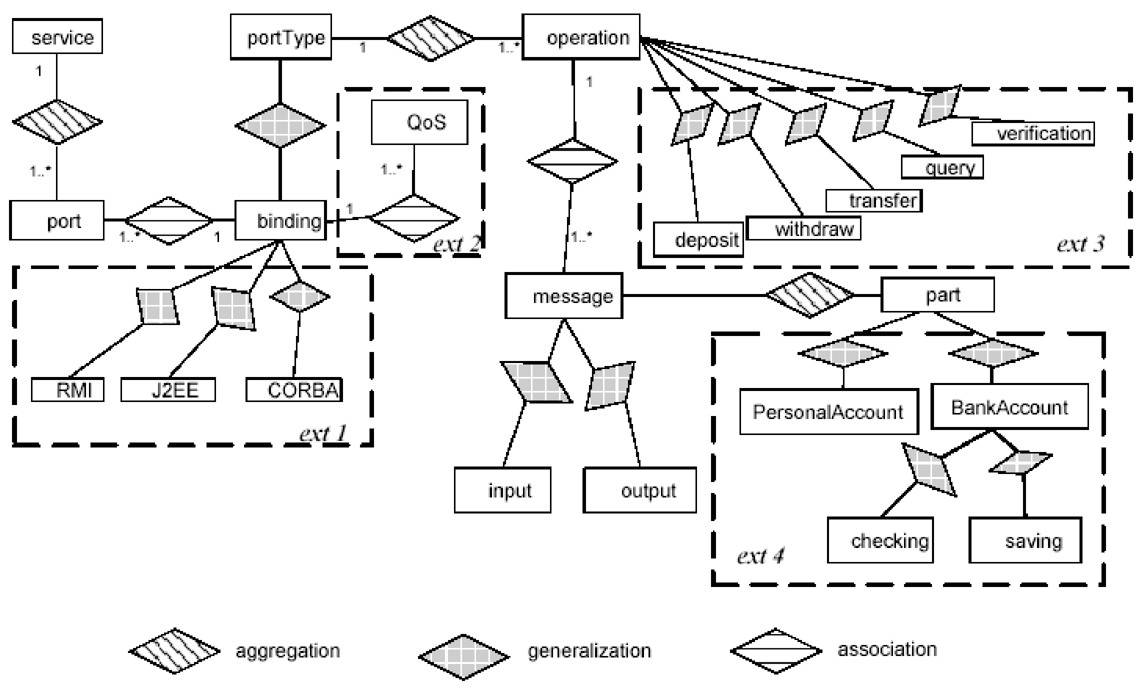


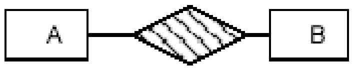
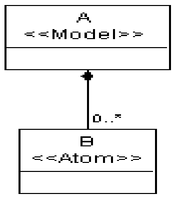
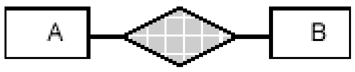
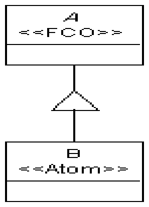
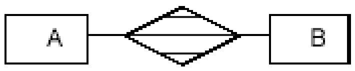
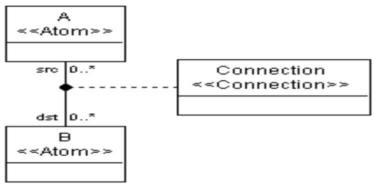
Figure 3.6: The ER model of Banking Service WSDL: the three parts enclosed with dashed line represent the extended part to the WSDL model.

3.3.2 Unmarshaling ER Model to GME Meta-model

In the GME meta-model, the containment relationship is represented by using a *model* element (stereotyped with <<*model*>>), which, in contrast to an *atom* element (stereotyped with <<*atom*>>), can contain other modeling elements. Also the contained elements can be promoted as *ports* of the model to have direct connections with external modeling elements. Additionally, GME uses a *root model* as an entry point of access to all the modeling elements. Also, the *relationship* of ER is represented in GME as a first-class modeling element, *connection* (stereotyped with <<*connection*>>), with a *connector* in the form of a dot to associate this relationship with two modeling elements (entities).

The unmarshaling from the ER model to the GME meta-model is based on the relationships in the ER representation, as is illustrated in Table 3.2.

Table 3.2: Unmarshaling rules: the relation notation is consistent with that in Figure 3.5

Rule Number	*Relationship type*	*GME Metamodel element*
1		
2		
3		 A <<Atom>> src 0..* Connection <<Connection>> dst 0..* B <<Atom>>

1). *A contains B***.** In this case, A can be modeled as a *model* element in GME containing B.

2). *B is specialized from A*. In this case, A is rendered by an abstract FCO (First-Class Object, tagged with <<*FCO*>>, represents an abstract generalization of other modeling constructs), a modeling element to be used as an abstract interface in GME, and B is represented as an inherited class of that FCO. Note there are two special treatments here: first, for the input/output elements of Figure 3.6, they are only used to tag the *connection* (named either "input" or "output") between message entities and its interconnecting entities in GME; second, the generalization relationship between binding and portType is actually treated as an association when modeling in GME, because the binding entity actually attaches values of the chosen protocol to the portType in WSDL rather than in the real sense of inheritance.

3). *B is associated to A*. In this case, a *connection* can be added to be associated with the A and B representations in GME. The connection element can be named with respect to A's or B's properties as a kind of tag, e.g., the tag can be named as the combination of both A's name and B's name. Note when the situation as described in case 2 applies, then this tag should be named as in case 2.

Figure 3.7 shows the meta-model created by unmarshaling the ER model in Figure 3.6 strictly observing the above unmarshaling rules. The seven boxes with bold borders correspond to the seven WSDL entities in Figure 3.5 and 3.6, with *WebService* corresponding to the *service* entity. The boxes in Figure 3.7 also contain attributes for the related models to be instantiated in the modeling phase. The four areas designated by four bold dashed circular lines correspond (from right to left) to the extension parts 1-4 in Figure 3.7. It can be seen from Figure 3.7 that the meta-modeling language lacks the modularity that programming languages have, thus the construction process of a complex

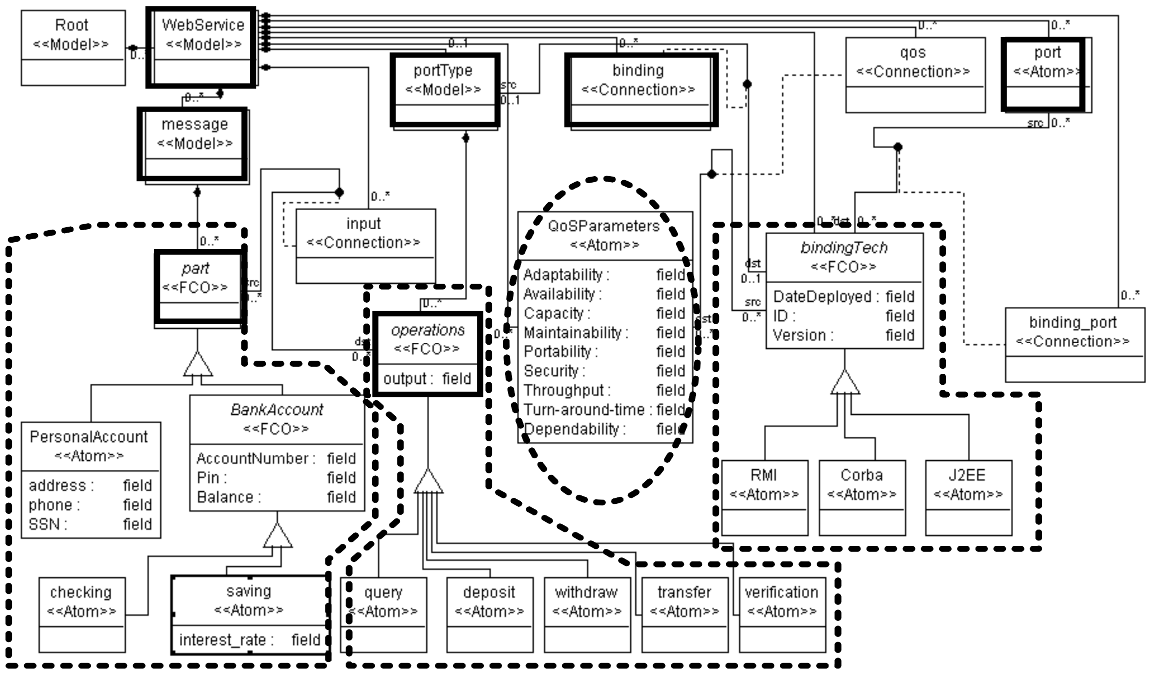
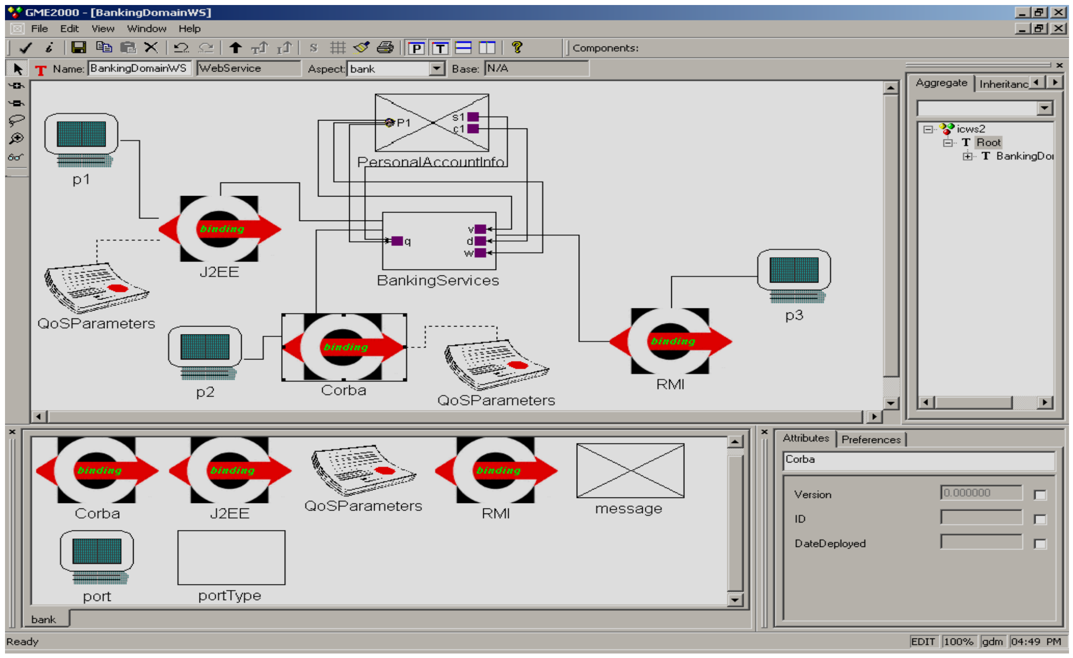


Figure 3.7: The meta-model of banking domain WSDL in GME

meta-model is error-prone without a systematic, automatable treatment.

3.4 The Web Services Modeling Environment

After a meta-model is derived by marshaling and unmarshaling models, a domain specific modeling environment (which is also a crucial part of MIC) can be created based upon the meta-model, as is indicated in Table 2.1. Figure 3.8 shows the screenshot of the banking-domain WS modeling environment based on the meta-model illustrated in Figure 3.7. The lower-left corner provides the modeling elements that can be dragged and dropped in the upper-left pane for constructing a banking service model. The names of the models in the lower-left pane represent the meta-model names (*kind names*); when



Name: BankingDomainWS WebService Aspect: bank Base: N/A

Corba

Version 0.000000
ID
DateDeployed

Figure 3.8: The banking domain-specific WS modeling environment

those models are dragged to the above pane, the model name can be changed to reflect the meaning of the model in the domain-specific context, which we call a *context name*. Furthermore, the domain-specific model can be traversed based on the meta-model and interpreted in terms of code generation using the GME Builder Object Network (BON) framework [ISIS, 01], which is illustrated in Figure 3.9. For saving space, Figure 3.9 only shows the interpreter code for generating the message and portType of WSDL. Other part of WSDL can be generated in a similar way. A snippet of the WSDL code generated for the banking service embedded with the QoS parameter extension is shown in Figure 3.10. Notice the bold-font part of the WSDL code in Figure 3.10 includes the QoS and ontology attributes of WSDL, which may be used for WS filtering if QoS requirements or domain specific requirements are included for service discovery.

```
const CBuilderModelList *root
       = builder.GetRootFolder()->GetRootModels();
POSITION pos = root->GetHeadPosition();
ASSERT(pos->GetCount()==1);
  //to ensure this model is representing just one WSDL

CBuilderModel *webserv = pos->GetHead();
  //get the handle to the WebService model
ASSERT(webserv->GetKindName()=="WebService");

//WSDL message part
const CBuilderAtomList *messages = webserv->GetModels("message");
pos=messages->GetHeadPosition();
CBuilderAtom *oneMessage;
while(pos)
  {
    /*
     traverse each message model and generating code
     <message>... </message>
     for each message model
    */

     oneMessage=messages->GetNext(pos);
     const CBuilderAtomList *accounts
             =oneMessage->GetAtoms("PersonalAccount");
    ...
   }

//WSDL portType part
const CBuilderAtomList *portType = webserv->GetModels("portType");
pos=portType->GetHeadPosition();
ASSERT(pos->GetCount()==1);
  //to ensure only one portType element in WSDL
CBuilderAtom *oneportType;
oneportType=portType->GetNext(pos);
…..
}
```

Figure 3.9: WSDL code synthesis using GME BON API

```
<message name="checking">
<part name="user_ident" type="identity"/>
<part name="p1" type="checking"/>
</message>

<message name="savings">
<part name="user_ident" type="identity"/>
<part name="p1" type="savings"/>
</message>

<message name="checking_savings">
<part name="user_ident" type="identity"/>
<part name="p1" type="checking"/>
<part name="p2" type="savings"/>
</message>

<portType name="BankingServices">
   <operation name="w"
          ontology="Banking:withdraw">
      <input message="checking"/>
      <output message=""/>
   </operation>

   <operation name="d"
          ontology="Banking:deposit">
      <input message="checking"/>
      <output message=""/>
   </operation>

   <operation name="v"
          ontology="Banking:deposit">
      <input message="checking_savings"/>
      <output message=""/>
   </operation>

   <operation name="q"
          ontology="Banking:query">
      <input message="savings"/>
      <output message=""/>
   </operation>
</portType>
```

(to be continued in the right pane)

```
<binding name="J2EE_Banking"
               type="BankingServices">
  <soap:binding style="J2EE"
     transport="http"
     QoS:portability="0.544400">
    .........
</binding>

<binding name="CORBA_Banking"
          type="BankingServices">
  <soap:binding style="CORBA"
     transport="IIOP"
     QoS:turn-around-time="10.35">
    .........
</binding>

<binding name="RMI_Banking"
          type="BankingServices">
  <soap:binding style="RMI"
     transport="http"
     QoS:dependability="0.34">
    .........
</binding>

<service name="My Bank">
  <port name="p1"
     binding="J2EE_Banking">
     <soap:address location="URL1"/>
  </port>

  <port name="p2"
     binding="CORBA_Banking">
     <soap:address location="URL2"/>
  </port>

  <port name="p3"
     binding="RMI_Banking">
     <soap:address location="URL3"/>
  </port>
</service>
```

Figure 3.10: The WSDL for a banking WS

3.5 Model Driven Approach to Enrich Web Services Semantics

Current WS standards mainly embrace the semantics of processes at the collaborating syntactic interface level. WSDL only exposes distributed object services, while such process behavior aspects as ordering, and dependency are not well specified in the existing WSDL standard. The model-driven approach can play a unique role in enriching the WS semantics:

- OCL (Object Constraint Language)[30] to enrich WS semantics at a high level

  OCL is used to complement the semantic representation for UML. Likewise, when the model is used to represent WS, OCL can be used to enrich WS semantics indirectly at a higher level. For example, if we add into the banking case in Figure 3.7 such requirement that "deposit and withdraw can only be applied to checking account", the specified constraints over withdraw and deposit operations can be enforced in GME using the following MCL expression [ISIS, 01], an OCL implementation in GME:

  ```
  connectedFCOs("src")->
          forAll(c|c. kindName()="checking")
  ```

  Those constraints apply to both the withdraw atom and the deposit atom in Figure 3.7, which means those First Class Objects (referring to both entities and relations in GME) that are connected with withdraw/deposit atoms are all of kind "checking". Therefore, in the WS modeling environment as shown in Figure 3.8, once a modeling entity of type other than "checking" is connected to withdraw/deposit, an error message window will pop up.

- Meta-model as Ontology

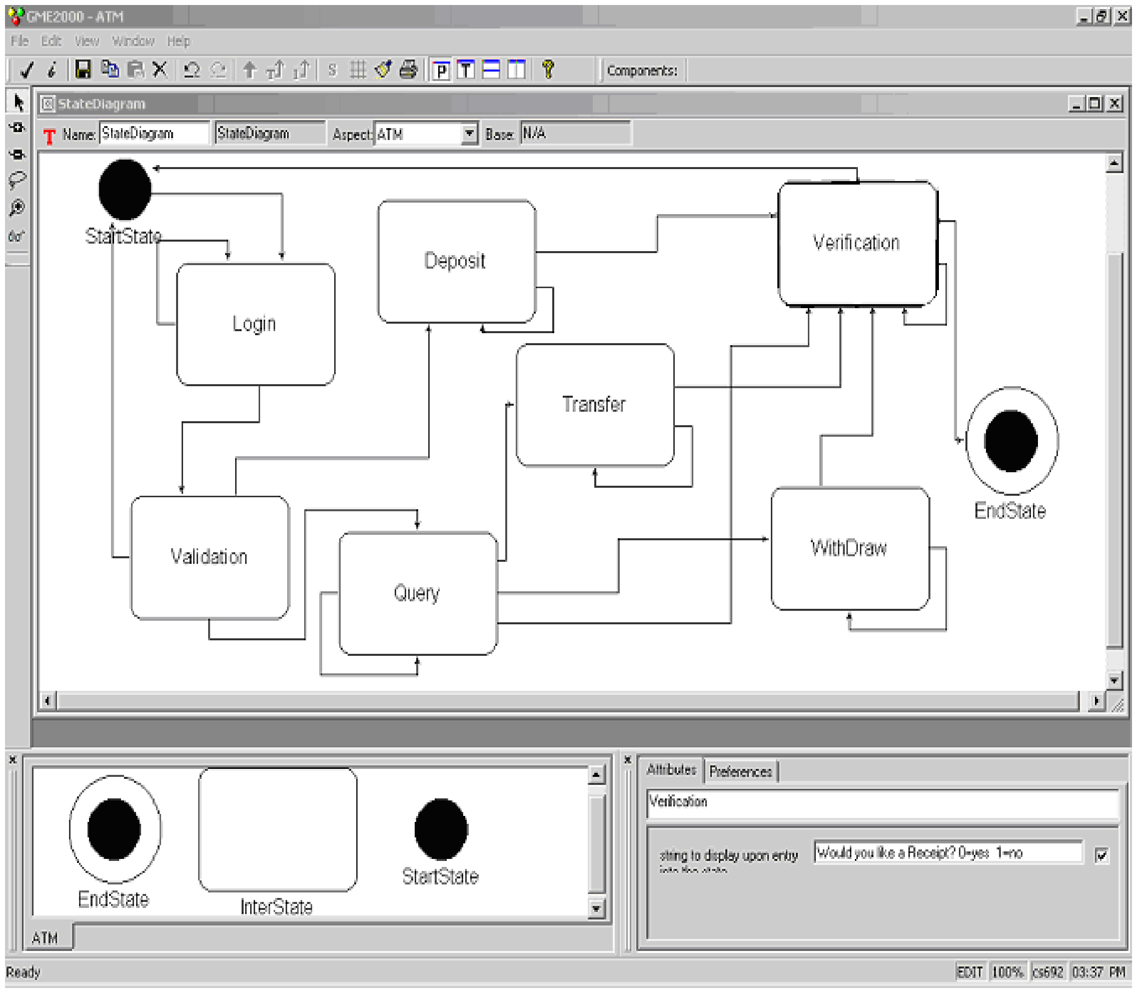
  A valid meta-model is an ontology, but not all ontologies are modeled explicitly as meta-models [Ernst, 02]. This ideal has already been used in [Hausmann, 04] for WS discovery. Comparatively, here we just output the meta-model informa-

[30] http://www-3.ibm.com/software/ad/library/standards/ocl.html

tion into the generated WSDL as ontology annotation to enrich the WSDL semantic representation.

- Creating modeling language for enriching WS semantics

  Assume there is an order restriction for those banking operations described in Figure 3.7: both *transfer* and *withdraw* have to be preceded by a *query* operation; the *account verification* comes after each of the other operations. Such models as Finite State Machine (FSM) can be used to enrich WS semantics. Based on the FSM meta-model as shown in Figure 2.6, a FSM modeling environment can be created in addition to the WS modeling environment of the preceding section, as shown in Figure 3.11. This environment can be used to generate operation ordering constraint code to be embedded in WSDL as shown in Figure 3.12. Note in the generated state transition code in Figure 3.12, the *condition* attributes are supposed to be customized in the specific banking behavior model before code generation, which for the sake of brevity are left empty here. The state transition specification generated here may be used in guiding the WS consumption and composition.



Name: StateDiagram StateDiagram Aspect: ATM Base: N/A

Verification

string to display upon entry into the state: Would you like a Receipt? 0=yes 1=no

Ready EDIT 100% cs692 03:37 PM

Figure 3.11. Banking behavior model based on FSM meta-model

```
<state>
   <state name= "Login" >
   <state name="Validation" >
   <state name="Query" >
   <state name="Deposit" >
   <state name="Transfer" >
   <state name="Withdraw" >
   <state name="Verification" >
</state>
<transition>
  <transition src="StartState"
   dst="Login" condition="">
  <transition src="Login" dst="Login"
   condition="">
  <transition src="Login"
   dst="Validation" condition="">
  <transition src="Validation"
    dst="Deposit" condition="">
  <transition src="Validation"
   dst="Query" condition="">
  <transition src="Deposit" dst="Deposit"
   condition="">
```

(to be continued in the right pane)

```
<transition src="Deposit"
    dst="Verification" condition="">
  <transition src="Query"
     dst="Transfer"  condition="">
  <transition src="Query"
     dst="Query"     condition="">
  <transition src="Query"
     dst="Withdraw"  condition="">
  <transition src="Query"
     dst="Verification" condition="">
  <transition src="Transfer"
     dst="Transfer" condition="">
  <transition src="Transfer"
     dst="Verification" condition="">
  <transition src="Verification"
     dst="StartState" condition="">
  <transition src="Verification"
     dst="Verification" condition="">
  <transition src="Verification"
     dst="EndState" condition="">
  <transition src="WithDraw"
     dst="WithDraw"  condition="">
  <transition src="WithDraw"
     dst="Verification" condition="">
```

Figure 3.12: Banking behavior model based on FSM meta-model

3.6 Related Work

This chapter presents both a novel model-driven approach in general and its novel application to WS in particular. As such, the related work comes twofold.

3.6.1 Model-driven approach

For the model-driven approach aspect, we use the ER model for marshaling and un-marshaling models. The related work in this regard includes:

- MDA

  MDA can assist in reengineering legacy software systems into PIMs. A PIM can be mapped to software components on PSMs, such as CORBA, J2EE or .NET. In this way, legacy systems can be reintegrated into new platforms efficiently and cost-effectively [Frankel, 2003]. However, the core part of mapping technology for MDA is either ad-hoc or pre-mature before MDA can be fully adopted in industry. ER-based model marshaling and unmarshaling offers a potential solution to address this problem systematically. Another difference is that in MDA, the PIM is treated as the dominant model while here we treat the technology domain as the dominant model, with business domain knowledge (PIM) as adjunct model in Section 3.

  It has been observed that the ER representation has been adopted in defining the Knowledge Discovery Meta-Model (KDM)[31] and Ontology Definition Meta-Model (ODM)[32] in OMG, which underscores the role that ER plays for model marshaling and unmarshaling.

- Grammar Inference

  The ER model, because of its powerful modeling capacity, can be used as an intermediate form for model-to-model and meta-model-to-meta-model exchange. Because of the dual role that the ER model can play, it is treated as an intermediate form for model-to-meta-model elicitation, which is the theme of this dissertation. This idea is very similar to grammar inference [Higuera, 2001], where a

[31] http://www.omg.org/cgi-bin/doc?lt/2003-11-4

[32] http://codip.grci.com/odm/draft/submission_text/ODMPrelimSubAug04R1.pdf

grammar can be inferred from language examples. But the two approaches are applied at different abstraction levels.

- XMI

  XMI[33] provides a standard mapping from MOF-based models to XML, which can be exchanged between software applications and tools, and the XMI specification is difficult to read by humans. In contrast, ER-based model marshaling and un-marshaling represents a design-level approach for evolving design assets, without being restricted to low-level syntactical data representation specifics, and the ER representation is much more human comprehensible. Also, the XMI-based approach uses top-down mapping, and is coupled to the meta-model of the targeted language; interchange format cannot be changed without changing the meta-model. In contrast, the ER-based approach represents either horizontal mapping or bottom-up mapping as is illustrated in Figure 3.1, without being tied to any meta-model.

3.6.2 Modeling WS

We applied the model-driven approach to WS, specifically, MIC for WS code generation automatically; Model-driven approaches for enriching WS semantics are also identified.

In [Lopes, 03], MDA is used together with workflow technology for modeling and composing WS. But the authors do not provide a guideline as to how to create the meta-models. Also the mapping from PIM to PSM is not detailed. In contrast, our meta-modeling approach is sufficiently complete and general as to be applicable to other aspects of WS

[33] XMI - XML Metadata Interchange - http://www.omg.org/technology/ documents/formal/xmi.htm

such as WS orchestration code generation. [Sivashanmugam, 03] describes an approach of adding semantics to WS by adding ontology attributes to both WSDL and UDDI, which includes pre-condition and effect specification. We applied ontology annotation to WS as well, and we put the pre-condition and other effect specification at the meta-model level. In [Mantell, 03], an MDA approach is used for BPEL4WS code generation from a UML design. This approach uses XMI processing technology for UML model exchange. Comparatively, the XML representation for the ER model is much simpler and easier to process in our approach. Code generation in [Mantell, 03] is based on the UML profile mapping, which is not as flexible as a generator-based approach in our case.

The UniFrame project ([Raje, 02]; [Olson, 05]) has a more comprehensive application of the model-driven approach. UniFrame aims at creating a framework for seamless integration of distributed heterogeneous components. In UniFrame, the model-driven approach is applied for domain engineering, and for creation of Generative Domain Models (GDMs) (Czarnecki, 2000), which are used for eliciting rules to generate glue/wrapper code for assembling distributed heterogeneous components. In contrast, the scope of glue/wrapper code generated here is specific to WS code, which has not been addressed by UniFrame.

3.7 Summary

With Web Services (WS) as a wrapper, legacy software systems can be reused and integrated beyond enterprise boundaries across heterogeneous platforms. This chapter has explored in detail a model-driven approach to reengineer legacy software systems to WS applications using a systematic, automatable process, which includes: 1) the meta-modeling process using ER-based marshaling and unmarshaling, 2) the construction of a WS modeling environment for generating WS code and enriching WS semantics. To our best knowledge, there is no peer work that addresses either systematic meta-model construction, or sufficient model-based WS code generation, while our work represents a comprehensive solution to both issues. Even though the work presented in this chapter is specific to WS development, the approach can be applied to other software system engineering by reengineering to a different meta-model other than the WS meta-model.

CHAPTER 4

DYNAMIC WEB SERVICES COMPOSITION

This chapter begins with the motivation for the dynamic WS composition, then presents two types of dynamic composition for distributed components: assertive and autonomous over .NET based Web Services environment. Case studies are provided to illustrate at a low level how the underlying infrastructure enables the dynamic composition, and to illustrate at a high level how dynamic compositions are specified.

4.1 Motivation

Revisiting the features of distributed components as described in Chapter 1,WS offers an interoperability infrastructure for distributed components as well as incorporating service discovery infrastructure in accordance with SOA. With problem (a) and (b) being embraced, current WS technology is yet to address the concerns as set forth in (c) and (d). Specifically,

1). *Service Provisioning*: in mission critical scenarios such as finance or military, there is a need for *guarantee of service availability* continuously, rather than shutting down the system for services adaptation;

2). *Service Consumption*: in distributed environments, service consumption experiences are dynamic and desirable to be seamless, thus the dynamic customizability of service s of vital importance in a service-oriented environment.

As such, static component composition is not adequate, and both functional and non-functional property adaptation need to be applied in a dynamic fashion. Along this line, this chapter presents a dynamic composition of WS for adapting WS functionally and non-functionally while maintaining the availability of WS. Here the WS environment is based on .NET CLR. We chose .NET because it is a thorough, fundamental re-architecting of a distributed computing platform based on WS, while other application server support for Web Services tends to be designed more as another client, or presentation tier for the back-end systems, with the communication tier based on RMI[34] or RMI/IIOP[35] rather than a strictly XML protocol based such as .NET [Newcomer, 02].

4.2 Overview of the Approach

4.2.1 Runtime code manipulation through assertive and autonomous composition rules

Figure 4.1 provides an overview of the dynamic composition approach. In the left pane of the *execution unit*, the .NET XML WS, which is specified with WSDL, is a layer built on top of .NET applications (1), which in turn runs over CLR (2). Consequently, .NET based XML WS can leverage the benefits of managed execution, where the .NET application is captured in the form of CIL (2), which is to be Just-In-Time (JIT) compiled into native code and executed (3). Therefore, by manipulating CIL derived from the XML WS implementation language, WS components can be composed at runtime.

The manipulation of CIL is illustrated in the right pane of the *configuration unit*, which is comprised of a stack of composition rules with a meta-level hierarchy. Composition rules are specifications for component composition (d). Meta-rules are specifications

[34] RMI – Remote Method Invocation: http://java.sun.com/rmi
[35] IIOP – Internet Inter-Orb Protocol

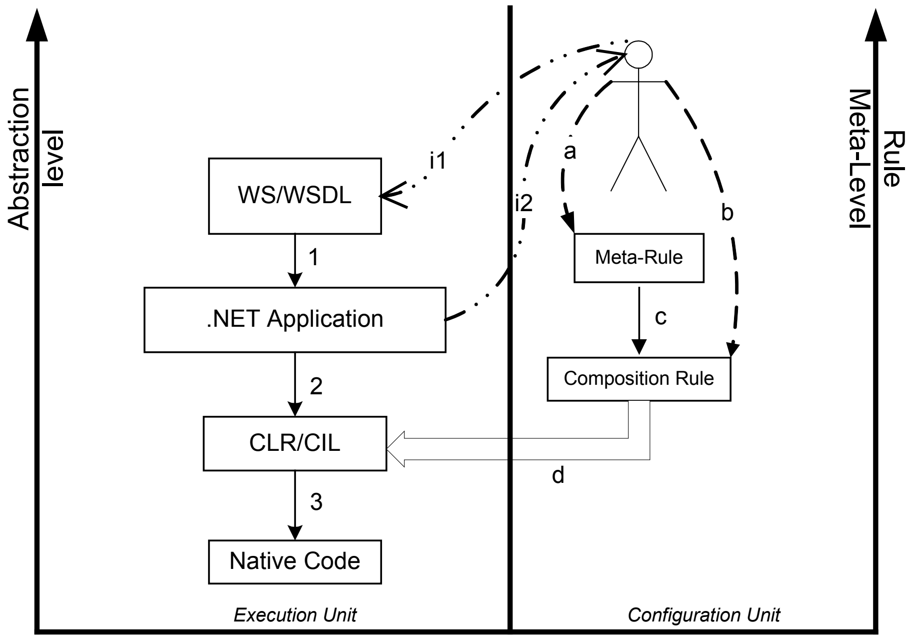


Figure 4.1: Overview of the dynamic composition approach

of triggering conditions for applying the composition rules, and the firing of the composition rules is enabled through a rule execution engine automatically (c). The use of rule engine for applying composition rules is useful for implementing *autonomous compositions* based on the runtime status quo. The actor icon represents a configuration console in a manual manner for both meta-rules (a) and composition rules (b). While the composition enabled through path (a->c->d) represents autonomous composition, the composition path of (b->d) represents the *assertive composition*. The configuration decision is based on WSDL exposed by WS (i1); WS itself can in turn assume the configuration role for specifying component composition reactively (i2).

4.2.2 Salient features

The dynamic component composition approach also includes the following salient features:

1). Non-invasive

- Non-invasive to application code for separation of composition concerns The WS composition is realized through in-memory IL manipulation as opposed to off-line invasive code change [Aßmann, 03]. The non-invasive change is often desirable as a WS vendor may deliver the software package in binary form. Also even though it is possible to derive CIL from a .NET executable using some de-compilation tools, invasively changing either original source code or derived CIL code will require unloading, recompiling and redeployment of the original WS application, which compromises the availability of WS. Moreover, the invasive change of WS code will pollute the original application such that recovering it will become difficult, which introduces the common version control problems for software systems.
- Non-invasive to platform for portability. The composition through manipulation of CIL at runtime (Figure 4.1-d) requires the interception of the managed execution. Instead of re-implementing the CLR such as rewriting open source CLR Rotor [Stutz, 03] to invasively add a listener for execution interception at the compromise of portability of CLR, we use a pluggable, configurable CLR profiling interface [Microsoft, 02] to achieve this goal, which can be enabled and disabled based on composition needs with ease to reduce unnecessary overhead.

2). Language neutral for cross-language component composition

By specifying composition rules based on WSDL, which in turn is based on a language neutral XML schema[36] , and code manipulation at the intermediate code (CIL) level based on language neutral CTS, WS components implemented in different .NET languages can be composed across language boundaries.

3). Adaptable composition.

With the configuration unit as a separate entity applied to runtime as shown in Figure 4.1, not only is the composition concern separated, but also it can be updated to realize adaptable composition at runtime.

The following section presents in detail the design and implementation of the dynamic component composition in Peer-to-Peer (P2P) scenarios, particularly, how the composition rules are specified to facilitate assertive and autonomous configuration.

4.3 Design and Implementation of Dynamic Web Services Composition

This section first introduces the composition paradigm for the dynamic WS composition scenario in Section 4.3.1, then describes the underlying enabling infrastructure for the dynamic WS composition in Section 4.3.2. In Section 4.3.3, the up-level programming model is introduced for the assertive dynamic WS composition, followed by descriptions of the justifications of the need of *Adaptation Advice Repository* (AAR) in Section 4.3.4. Section 4.3.5 complements Section 4.3.3 by presenting the autonomous dynamic WS composition, with the need of the rule inference engine justified.

[36]http://www.w3c.org/2001/XMLSchema

4.3.1 Composition in Peer-to-Peer (P2P) paradigm

Figure 4.2 illustrates the architecture for the dynamic WS component composition based on .NET WS environment. In our work, each WS component is hosted in an infrastructure *DynaCom*, which is essentially a profiler-enabled CLR to be detailed in Section 4.3.3. DynaCom is used as a proxy for WS to interoperate with components in other locations through WS. Meanwhile, DynaCom can intercept the execution of the hosting components and change the behaviour of the executing WS dynamically. DynaCom is based on our prior work on using a profiling approach for dynamic service provisioning [Cao-c, 05], but here it is tailored to WS composition, which has been initially given a detailed account in [Cao-e, 05].

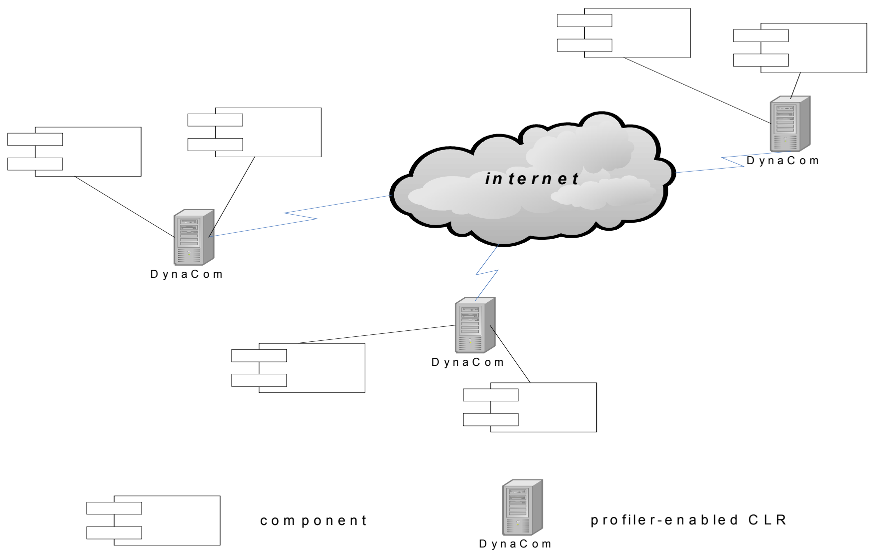


Figure 4.2: The P2P WS component compositions in .NET WS environment

The composition model shown in Figure 4.2 represents a P2P paradigm, which is the primary composition model to be addressed in this chapter. This choice is based on the observations that P2P and dynamic composition are tightly associated:

1). *P2P as an agile mode to accommodate dynamic features.* While WS orchestration by executing BPEL4WS[37] in the execution engine represents a centralized composition model, it has been observed that such a composition model compromises scalability, availability, and security for the server [Chen, 01]. With the highly dynamic features of a distributed environment, the P2P component composition paradigm will be more widely used.

2). *Dynamic composition is the necessary means for realizing P2P computation in a distributed environment*. While component composition usually requires the generation of glue/wrapper code [Cao-b, 02], the physical location for hosting the generated glue/wrapper code is a hard problem in P2P mode without central management and storage units. Dynamic composition, with glue/wrapper code generated in memory and JIT compiled and executed at runtime, provides a solution for P2P component composition without the physical code placement issues.

4.3.2 Infrastructure for WS composition

DynaCom is the enabling infrastructure for dynamic WS composition. Figure 4.3 provides an anatomy of DynaCom. The part enclosed by the big square represents the enabling mechanism for dynamic composition, which is transparent to the components to be composed above the big square.

[37] BPEL4WS - Business Process Execution Language for Web Services - http://www-128.ibm.com/developerworks /library /specification/ws-bpel

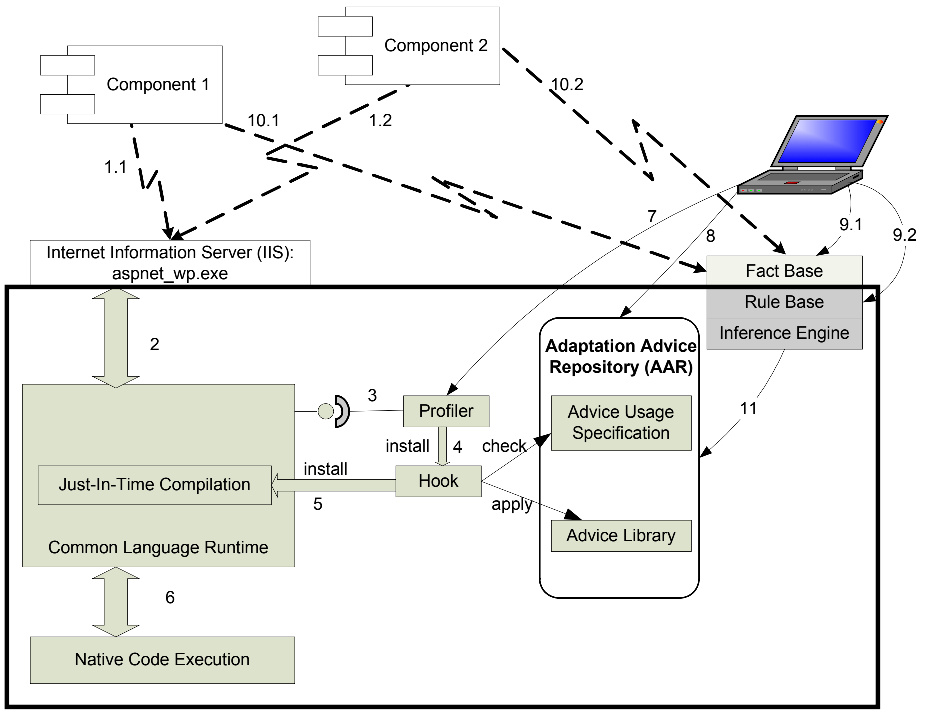


Figure 4.3: The architecture of **DynaCom: Dyna**mic **Com**position enabling unit, which includes the part enclosed by a bold-border rectangular and the IIS, facts. The parts of IIS and facts are accessible to the remote components, while the enclosed part of DynaCom are only accessible locally. The dashed line of 1 and 10 represents remote access, while all the remaining solid lines represent local access. The laptop icon represents the local configuration unit to DynaCom.

Our work is built upon the ASP.NET[38], a WS implementation package based on the .NET framework. In ASP.NET, Internet Information Service (IIS)[39] is used to accept the incoming WS SOAP (Simple Object Access Protocol) [Newcomer, 02] message transported over HTTP (1) in Figure 4.3. Upon acceptance of the WS request encoded as a SOAP message, an IIS filter will launch a work process (aspnet_wp.exe), which in turn will launch CLR (2) to run the WS application in the mode of managed execution. At this

[38] http://asp.net
[39] http://www.microsoft.com/WindowsServer2003/iis/default.mspx

point, the WS application is rendered as in CIL subject to be JITcompiled into native code and executed (6). In order to adapt WS, it is needed to intercept the WS call at the CIL level before it is compiled. While it is reasonable to implement the expected functionalities in the CLR open source of millions of lines of code such as Rotor [Stutz, 03], we feel it too expensive an effort. Instead, we use the CLR profiling API to implement a *Profiler* as event handlers, and register them as listeners for the events generated from the CLR (3). In contrast to the conventional publisher/listener model, which is often of a client-server relationship, the profiler here will be mapped into the same address space for the profiled application as an in-process server.

The events generated from the CLR are the result of managed execution, including but not limited to garbage collection, class loading/unloading, CLR startup/shutdown and JIT compilation. The event of our interest is JIT compilation, for which we implement in-memory CIL manipulation for the event handler. The adapted CIL will then be JIT compiled and executed resulting in changed WS behavior. A one-shot change to CIL will reduce the traceability of adaptation, impede the removal of the imposed adaptation (thus incapable of dynamic decomposition), and restrict the flexibility of further adaptation. Therefore, we interpose *Hook* code (4,5) in the WS application to be adapted, which will check the AAR for applicable adaptation advice. The term “advice” is further explained in the next section. AAR is located in a shared memory for fast access during in-memory CIL manipulation. The AAR includes an Advice Library storing predefined reusable advice in the compiled managed code form, as well as an *Aspect Usage Specification* (AUS) component to indicate applicable advice for WS. The Profiler and the AAR are subject to external configuration (7-11): for 7, the configuration is used to narrow down the scope of profiling; for 8-11, the configuration is used to dynamically specify adaptation rules, among which 8 corresponds to a direct manipulation of adaptation rules, while 9-11 corresponds to indirect manipulation of adaptation rules through a rule inference engine. The inference engine can dynamically inject AUS into AAR based on the rule specification, which is to be detailed in Section 4.3.5. The laptop icon in the upper-right corner represents the local configuration unit. The configuration unit for DynaCom can adopt a GUI interface or an API interface. In our work, we use a simple console for the local configuration unit handling configuration 7-9, while configuration 10-11 is realized through an API interface.

4.3.3 Programming model

4.3.3.1 AOP for WS composition specification

While the preceding section describes the underlying infrastructure, this section describes the up-level programming model for WS composition. In Figure 4.2, each DynaCom only hosts 2 components, which is for simplicity purpose in illustration. In reality, a DynaCom may be hosting multiple components. Consequently, a component handling a crosscutting concern may be expected to be composed with multiple other components. Thereafter, it is not possible to specify adaptation for every individual component upon changing of requirements. Instead, there needs to be a means to abstract the adaptation in a modularized way. AOP offers a means to abstract cross-cutting concerns in a modularized way called an *aspect*, and the concerns can be weaved using weaver technology into the base program based on the *join point model*, which specifies the destination to weave concerns. In the same vein, we specify the adaptation advice in the AAR in a modularized way following AOP style, and the composition specification is rendered as an aspect weaving specification.

Moreover, AOP also offers a means for separating composition specification from components to be composed, with the underlying weaver realizing the composition. As such, in case the components to be composed do not involve crosscutting concerns, the component composition is still specified in the same way as an aspect weaving specification with AUS.

The AOP weaving specification in AspectJ [Kiczales, 01] can be adapted for component composition specification in terms of aspect weaving as illustrated in Table 4.1. While the aspect weaving specification in Table 4.1 remains AspectJ-like, which facilitates programming, the actual aspect weaving specification is specified with XML-based AUS; the AspectJ-like specification can be translated into XML-based AUS.

4.3.3.2 Implementation of dynamic weaver

To weave and unweave the specified advice, we instrument the hooks at both the entry (*pre-hook*) and exit point (*post-hook*) of the WS method to be adapted, which are used to check into the AAR to see if corresponding *before advice* and *after advice* is applicable: the former performing some pre-processing before the actual WS method execution, while the latter performs some post-processing immediately before the WS method execution returns. Such pre- and post- processing capacity can be used to instrument codes for addressing non-functional concerns, such as applying access control upon the entry into the WS method, or applying state persistency service for the executed WS application upon the end of the WS call.

Table 4.1: Composition specification in the form of aspect weaving

	Component Composition	*Aspect Weaving Specification*
Sequential	a precedes b	after (a) {b; }
	a follows b	before (a) {b; }
Wrapping	a is wrapped by b at the beginning and c at the end	around (a) {b; proceed(); c; }
Overiding	a is overridden by b	around (a) {b;}

Also included in the pre-hook are the instructions to check if an *around advice* is specified or not, and a jump instruction to redirect the execution to the exit point of the WS application. The jump instruction is to be activated if an around advice is found valid in the AAR. With around advice, the original WS will be replaced with new behaviour specified in that around advice. Consequently, not only the original WS can be decorated, it can also be overridden completely, which is necessary when a buggy WS is identified and needs to be removed, or an old service module need to be updated. The around advice sufficiently offers a delegation and wrapping approach for component composition which is exemplified in Section 4.4. By using a hook for weaving, advice can be applied dynamically and proactively. Meanwhile, unweaving advice can be realized by dis-activation of the corresponding AUS in AAR. Figure 4.4 is the CIL manipulation template for adapting a WS method. An example is also provided in Appendix B to illustrate the code manipulation at the binary level.

```
IL_0000: ldstr "classname/method_name/parameter_name_list/returntype/before"
IL_0005: call    void dynaweave.hook::advising(string) //to check & apply before-advice
IL_000a: pop   //to maintain the original stack
IL_000b: ldstr "classname/method_name/parameter_name_list/returntype/around"
IL_0010: call  void dynaweave.hook::advising(string) //to check & apply around-advice
IL_0015: brtrue IL_020b
IL_001a:  <Original Method body in IL>
............
IL_0200: ldstr "classname/method_name/parameter_name_list/returntype/after"
IL_0205: call   bool dynaweave.hooker::advising(string) //to check & apply after-advice
IL_020a: pop //to recover the original stack after original method is executed
IL_020b: ret
```

Figure 4.4: Instrumentation of IL code of a WS method

Note in this section composition specification is assertively applied to the components to be composed based on the dynamic weaver, without the consideration of autonomy in the dynamic environment: that part is to be detailed in Section 4.3.5, with the description of the rule inference engine introduced, which is necessary for autonomous composition.

4.3.4 The need for a serializeable aspect weaving specification in XML

In DynaCom, while AUS can take the form of AspectJ-like syntax as shown in Table 4.1, AUS is subject to be updated in a dynamic environment, especially in the autonomous composition scenario as introduced in Section 4.3.5. As such, AUS needs to be represented in a serializeable manner in the persistent form of AAR. XML is widely used to serialize data between applications. We use the XML for AUS as well for serializeable aspect weaving specification, which also has the following underpinnings:

- As shown in Figure 4.3, the execution of .NET based WS applications are captured at the CLR level based on CTS, which is type system neutral to any .NET application language. Consequently, an aspect weaving specification based on CTS will be applicable to all .NET WS applications. However, writing adaptation AUS based on low level CTS is error-prone and not necessary for high-level AUS. Also, AUS, as the specification reflecting the business requirement adjustment (by composing and decomposing related components), should have an abstraction level close to business requirements, rather than being tied to underlying implementation details. On the other hand, WSDL is based on the XML Schema, which is another language neutral type system that can be mapped to the language-neutral CTS. The XML Schema based specification is parsed and translated to CTS to be matched against the string provided by the hook such as described in IL_0000, IL_000b, IL_0200 in Figure 4.4.
- Components delivered may be in binary form with source code being unavailable, thus AUS at the application code level is not feasible. On the other hand, components in the .NET WS environment are exposed through the WSDL interface, which offers a reference point for specifying WS component adaptation.

The XML schema for AUS is illustrated in Figure 4.5. Associated with each *advicename* is the path information for actual advice in the form of managed code stored

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="tns:advicetype"/>
  <wsdl:input message="tns:return_type"/>
  <wsdl:input message="tns:classname"/>
  <wsdl:input message="tns:methodname"/>
  <wsdl:input message="tns:parameter_list"/>
  <wsdl:input message="tns:advicename"/>
</wsdl:operation>
```

Figure 4.5: The AUS schema

in the AAR. All the advice code is defined as a template with the tuple <*Classname, Methodname, Parameter_List*> as parameters, which offers reusability of advice. Such advice can be pre-built in any .NET language and compiled into managed code. If a matching advice is found, then the advice code will be loaded from the corresponding path and called. In our work, the wild-card characters are also supported for AUS.

4.3.5 Autonomous component composition using a rule inference engine

4.3.5.1 The need for a rule inference engine

Functionality for the composed distributed software systems can be predicted based on the constituent components [Hissam, 03], thus a component composition based on functional requirements can be specified assertively. In contrast, non-functional properties such as pricing based on end-to-end delay (service consumption duration) for composed distributed software systems can only be reasoned about at runtime because of their dynamic characteristics. As such, a distributed software system needs to self-adapt itself by composing and decomposing components autonomously to achieve the expected QoS. While programmatically incorporating all adaptation decisions are theoretically sound, it is not practically feasible. Consequently, re-writing and recompiling the code upon changed adaptation decision are necessary, which is not appropriate for dynamic composition. Using an inference engine, the rules can be specified declaratively in a logic programming style, which can further be executed directly in an interpretive fashion, as opposed to being specified in an imperative fashion and need to be further compiled before execution. Therefore, with the capacity of maintaining the execution of runtime, inference-engine based composition rule specification aligns with the dynamic composition paradigm.

Moreover, the declarative rule specification is at an abstraction level closer to user requirements than the programming language, which is easier to be derived from user requirements. Also, with pattern matching and first-order logic, the declarative rule specification can be used to sufficiently specify the WS selection, which is incorporated as part of the WS composition rule specification to be executed by the rule inference engine seamlessly. This is to be exemplified further in Section 4.4.3.

4.3.5.2 Jess as the rule engine

In our work, we use Jess [Friedman-Hill, 05] as the underlying inference engine, which is a forward and backward chaining rule engine for the Java platform. Associated with the inference engine are the fact bases and the rule base as shown in Figure 4.3. The rule base is only accessible to the local hosting site, and represents local autonomous composition policies; comparatively, the fact base is exposed to both the local and remote site, which can be manipulated by either the local configuration unit, local components, or remote components. The fact bases of different DynaCom are federated, and a local rule engine can query a remote fact base for triggering an action. This is useful when a local composition rule is dependent on remote component status (which is reflected in the remote fact base). For example, the unavailability of remote components during a certain period of time will trigger the local component to connect to an alternative component, which offers a means of fault tolerance.

Jess offers a hybrid programming paradigm between the Java language and declarative rule specification: the Java code can invoke the Jess rule engine while the Jess rules invoke Java code. In order for the Jess fact base to be interoperate with remote components, as well as to enable the Java-based inference engine to be interoperable with the .NET environment, we wrap the Java-based Jess API with a WS layer using Java WSDP[40].

4.3.5.3 Rule specification for autonomous composition

The self-adaptation decisions can be collectively built into a knowledge base proactively and retroactively. Therefore, the complete dynamic component specification in terms of dynamic, autonomous aspect weaving rule takes the following form[41]:

```
 apply [aspect_name] when [logical_condition]
```

The corresponding Jess rule specification is:

[40] Java WSDP – Java Web Services Developer Pack – http://java.sun.com/ webservices/jwsdp/index.jsp
[41] Note this specification is semantically equivalent to the AUL introduced in Section 2.5.3.1.

```
(defrule aspect-weaving
 ([logical_condition in])
   =>(apply [aspect_name]))
```

The `when` clause represents the condition under which the action `apply [as-pect_name]` is to be performed, which in turn will add an AUS corresponding to . `apply [aspect_name]` into the AAR through Jess-.NET bridge to be detailed in Section 4.4.2.

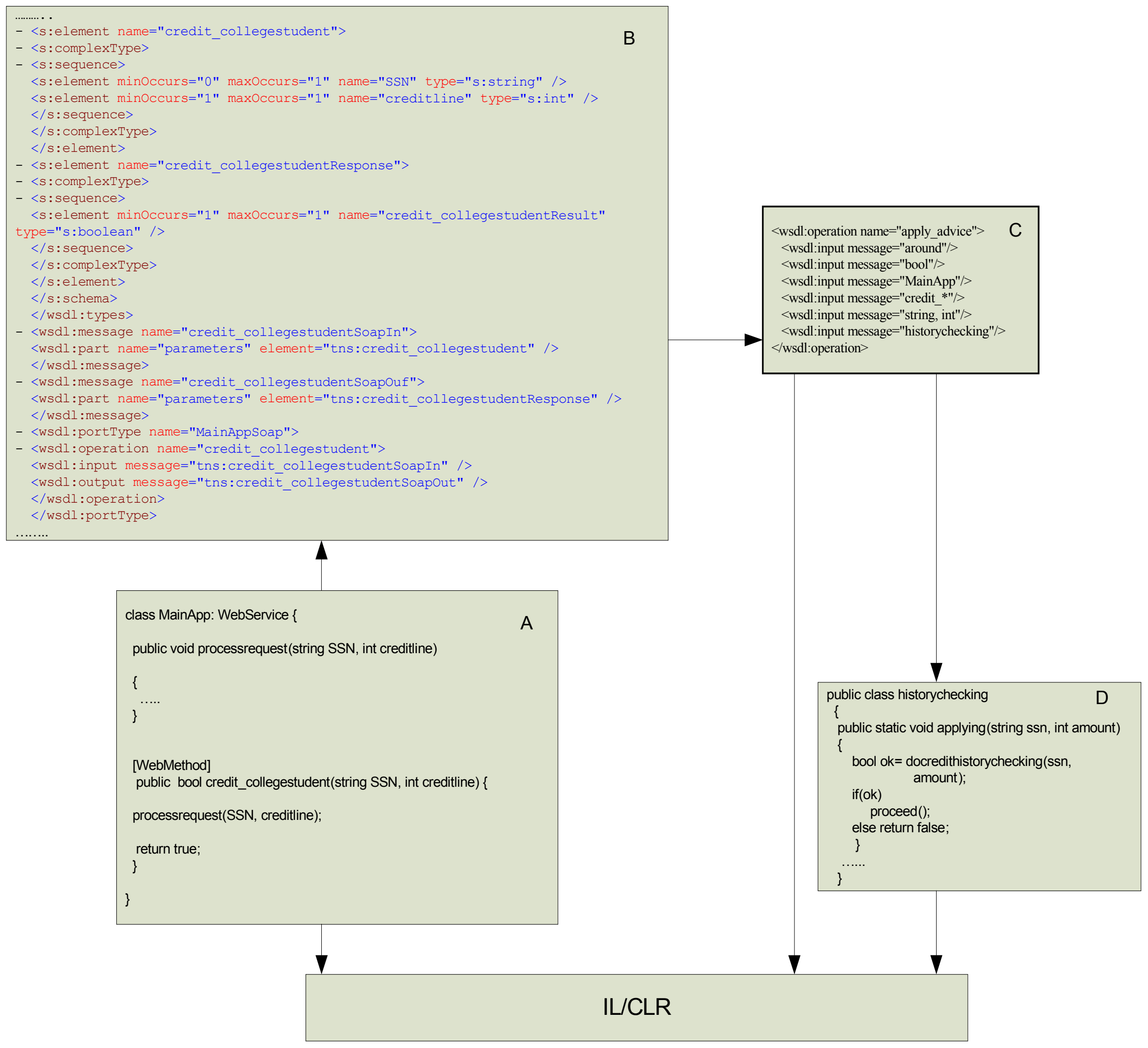
4.4 Case Study

In this section we present three case studies. They are complementary to each other in the sense that:

- The first one in Section 4.4.1 is an assertive dynamic composition example which is also intended as a shortcut to illustrate how the underlying infrastructural parts shown in Figure 4.3 (except the rule inference engine part) work together.
- The second one in Section 4.4.2 showcases up-level programming model of dynamic WS composition, particularly the the use of Jess language and its interoperation with .NET. for autonomous WS composition.
- The third one in Section 4.4.3 further demonstrates the power of declarative logic programming for the autonomous dynamic composition specification.

4.4.1 Composing crosscutting credit authorization WS components - putting the pieces together

Figure 4.6 provides an example of a college student credit authorization WS to demonstrate the assertive dynamic component composition for a non-functional concern:



```
………. .
- <s:element name="credit_collegestudent">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="SSN" type="s:string" />
  <s:element minOccurs="1" maxOccurs="1" name="creditline" type="s:int" />
  </s:sequence>
  </s:complexType>
  </s:element>
- <s:element name="credit_collegestudentResponse">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="credit_collegestudentResult"
type="s:boolean" />
  </s:sequence>
  </s:complexType>
  </s:element>
  </s:schema>
  </wsdl:types>
- <wsdl:message name="credit_collegestudentSoapIn">
  <wsdl:part name="parameters" element="tns:credit_collegestudent" />
  </wsdl:message>
- <wsdl:message name="credit_collegestudentSoapOuf">
  <wsdl:part name="parameters" element="tns:credit_collegestudentResponse" />
  </wsdl:message>
- <wsdl:portType name="MainAppSoap">
- <wsdl:operation name="credit_collegestudent">
  <wsdl:input message="tns:credit_collegestudentSoapIn" />
  <wsdl:output message="tns:credit_collegestudentSoapOut" />
  </wsdl:operation>
  </wsdl:portType>
……..
```

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="around"/>
  <wsdl:input message="bool"/>
  <wsdl:input message="MainApp"/>
  <wsdl:input message="credit_*"/>
  <wsdl:input message="string, int"/>
  <wsdl:input message="historychecking"/>
</wsdl:operation>
```

```
class MainApp: WebService {

  public void processrequest(string SSN, int creditline)

  {
   …..
  }


  [WebMethod]
   public  bool credit_collegestudent(string SSN, int creditline) {

  processrequest(SSN, creditline);

   return true;
  }

}
```

```
public class historychecking
  {
   public static void applying(string ssn, int amount)
   {
      bool ok= docredithistorychecking(ssn,
                     amount);
      if(ok)
          proceed();
      else return false;
       }
    ……
   }
```

Figure 4.6: Composing credit authorization component assertively

access control. Figure 4.6-A provides a simple WS application written in C#, which provides a WS method for authorizing credit card application based on the Social Security Number (SSN[42]) and the expected credit line. The corresponding WSDL in Figure 4.6-B can be automatically generated from the source code in Figure 4.6-A based in ASP.NET,

[42] An identification number used to identify income earners in the United States.

which in turn is to be exported and used as the basis for AUS as well. Figure 4.6-C is an AUS with around advice to apply credit history checking before any credit card application request is processed. The AUS represents a sequential composition specification for a component encapsulating crosscutting concerns (here *HistoryChecking*). The wild card specification in credit_* represents all credit application with the request name preceded with "credit_". Figure 4.6-D is the source code for the pre-built credit history checking advice, which can be written in any .NET language (here C#) and is compiled and persists in the managed code form. The type systems in Figure 4.6-A, Figure 4.6-C, Figure 4.6-D are translated into CIL and are matched up in CLR. Once a match holds, the advice in Figure 4.6-D will be called by the hook instrumented at runtime. The WS application source code level detail is transparent to AUS in Figure 4.6-C, as well as to the HistoryChecking component in Figure 4.6-D. By instrumentation of intermediate code, component composition can be realized across language boundaries without invasively changing application source code.

4.4.2 Composing travel planning WS components – dynamic composition programming model illustrated

The former section demonstrates how each part in DynaCom is integrated together for assertive dynamic component composition, particularly how the intermediate code manipulation enables the component composition across language boundaries without invasively accessing the application source code. This section will further explore the dynamic composition for multiple components for travel planning, which not only includes assertive dynamic composition, but also autonomous dynamic composition using the Jess rule inference engine. Complementing the previous case, this case focuses on the user level component composition specification as opposed to dwelling on the low level intermediate code manipulation.

In Figure 4.7, the boxed part contains the WS components for travel planning, with those above the box representing the types used in the WS components.

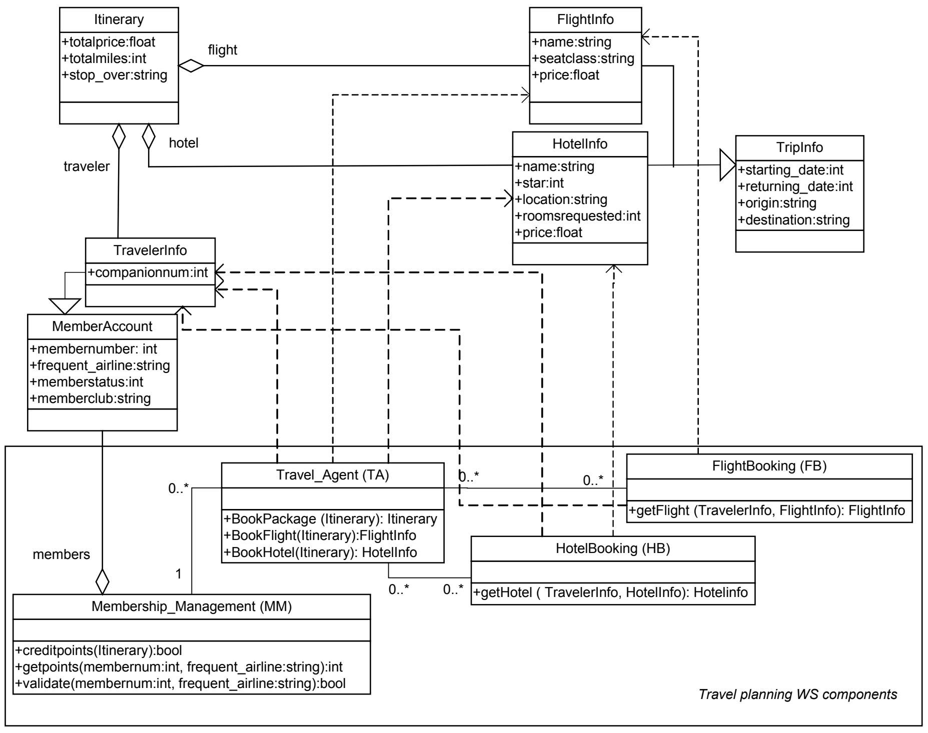


Figure 4.7: Class diagram for travel planning WS components

Each customer plans the travel through a travel agent *Travel_Agent (TA)*. The travel agent will handle both the booking of flight, *FlightBooking (FB)*, and hotel, *Hotel-Booking (HB)*. All travelers can credit their mileage into their own frequent flier number through *Membership_Management (MM)*. They can book the travel package including both hotel and flight, or just book one of them. They can also book for group travelers.

The result of the travel booking process is the itinerary information (*Itinerary*), which includes the total cost of the trip. All those WS components in the box are loosely coupled and dynamically bound based on their partnership, service charge, and QoS. Figure 4.8 illustrates the travelling components composition process with sequence diagram. The italicized part represents the dynamically composed components; the TA and its associated methods represents the static front end travel agent components to the customers with back end components dynamically composed on demand.

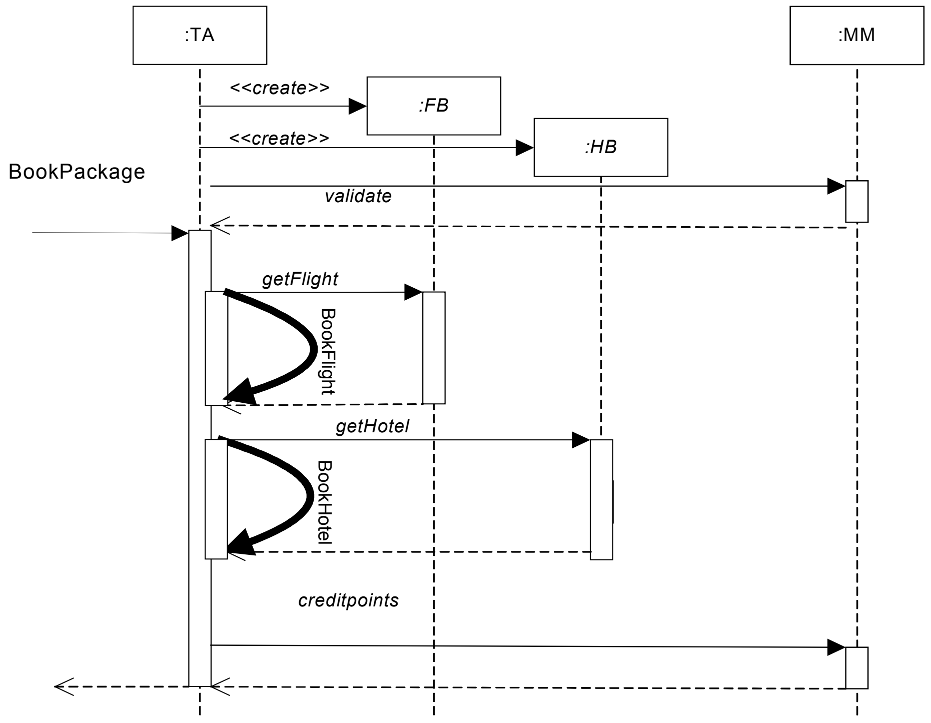


Figure 4.8: Dynamic composing travel planning WS components

4.4.2.1 Static front end

During travel planning, the customer starts from TA WS method *BookPackage*, with the backend components dynamically composed to fulfill the travel planning purpose. The TA serves as front end components to the customers to be dynamically bound to backend WS components, and the *BookTravel* method is implemented as shown below:

```
Itinerary BookPackage (Itinerary it)
 {
    FlightInfo fi;
     HotelInfo hi;
     fi=BookFlight (it);
     hi=BookHotel(it);
     return combine(it1,it2);
   }
```

4.4.2.2 Dynamic backend

While the front end code as shown above is static to the customer side, there are some dynamic component composition concerns in the backend that is transparent to the customers:

- Dynamic partnership

The front end TA component may have dynamic partnership with back end FB and HB based on their mutual contract, service charge (if the service charge is exceeding the budget, the partnership will be cancelled and a new partner will be identified), or QoS (if the service of the current partner is down, an alternative partner need to be identified). Note we assume membership management is centralized and statically bound in this case in accordance to the real world examples, where membership such as Social Security Account is centrally administrated by the appropriate government agency. As such, the partnership should be established dynamically, which, consequently, is also subject to dynamic change. Figure 4.8 illustrates the dynamic partnership establishment by using two <<*create*>> messages before the call of *BookPackage*, which can be translated into the following[43]:

```
before(Itinerary *.BookPackage (Itinerary it))
{

 this.fb= new FB(…);
//the "…"part provides the
//information referencing the
//actual FB component that
//the instantiated object is bound to

 this.hb= new HB(…);
}
```

Furthermore, the front end BookFlight and BookHotel code is dynamically overridden to delegate to the actual methods of FB and HB respectively. This is achieved using around advice as shown below:

```
around (FlightInfo *.BookFlight (Itinerary it))
 {
  return fb.getFlight (it.traveler, it.flight);
  }

around (HotelInfo *.BookHotel (Itinerary it))
 {
  return fb.getHotel (it.traveler, it.hotel);
     }
```

[43] For illustrative purpose, we use the syntax resembling AspectJ to specify the component composition, which in turn will be translated into XML representation as is described in Section 4.3.4.

- Dynamic membership management.

With the tightening security measures, the customer's background is subject to be checked by the central member management (MM) unit within a designated period of time. As such, a rule is added in Jess that for a given duration, the membership will be validated (e.g., background checking, passport verification) for each *BookPackage* call. Assume during the period July 1, 2005, to September 20, 2005, all traveller's membership will be validated by MM. To enable the Jess rule engine to trigger the dynamic composition of validation behavior, we need to:

1) capture the execution of *BookPackage* and relay the values into Jess fact bases;
2) have a bridge from Jess to .NET for rules to directly manipulate AAR in Figure 4.3.

As is mentioned in Section 4.3.5.2, we use WS to wrap a Java class, which in turn can interoperate with Jess. Thus, a .NET based WS component can interoperate with Jess rules. Specifically, to achieve 1), we add the following code into the "before advice" for BookPackage:

```
before(Itinerary *.BookPackage (Itinerary it))
{ …… //above are other advice code which are ignored
    //here for clarity
WS_Jess.assert("membernumber",
                it.traveler.membernumber  );
 WS_Jess.assert("airline",
               it.traveler.frequent_airline);
Date date=getdate();
WS_Jess.assert("date",date);
//the above three lines add three
//facts to the Jess fact base through WS-Jess bridge
}
```

To achieve 2), we define a Java class which is used as a relay between Jess and the.NET platform, so that whenever the rule fires, AAR in .NET can be manipulated from Jess. The Java class is defined as follows:

```
class Jess_WS{
 public static void
     apply(string advicetype, String  returntype,
           String classname, String methodname,
            String parameterlist, String advicename)
     {
      … //code to interoperate with .NET to update AAR;
     }
}
```

The parameter list is consistent with the XML elements as shown in Figure 4.5. The Jess rule is specified as follows, which calls into the Java class Jess_WS:

```
(bind ?aus (new Jess_WS)) ;;aus_wrapper is the Java
       ;;wrapper for writing AUS
         ;;into the AAR through Java-WS bridge using
       ;;Java WSDP as ;;described in Section 4.3.5.2
(defrule security_control
(date ?d &:(>= ?d 20050701)&:(< ?d 20050920))
  =>(?aus  apply "before", "", "TA", "BookFlight", "",
  "MM.validate"))
```

The last line defines a Jess rule specifying once the booking date is between July 1, 2005 and September 20, 2005, the membership validation advice will be applied through Jess-Java-WS interoperation before the call of *.BookFlight in the .NET environment. Once the condition is satisfied during runtime, the corresponding rule will be applied autonomously for dynamic composition. Furthermore, as the Jess rule exists as a separate entity for configuration from the execution logic, the composition rule can be adapted as needed at runtime as well.

Likewise, dynamic composition can be applied to credit travel points after the travel reservation, using after advice:

```
after(Itinerary *.BookPackage (Itinerary it))
{ MM.creditpoints(it);
}
```

Furthermore, dynamic composition can be applied either assertively or autonomously as shown above for other non-functional property guarantees including but not limited to budgeting (if the cost of the requested service exceeds the budget, either to choose a cheaper service or to remove subcomponents for reducing cost), and load balancing (if current load is over capacity, the service requests are to be delegated to alternative components). As those composition specifications overlap the aforementioned dynamic composition specifications in principle, they are ignored here to avoid duplication.

4.4.3 A financial WS portal: composition specification through declarative logic programming

This section demonstrates the power of declarative logic programming for specifying WS composition. In particular, this section will show how the gap between composition requirements and the execution of the composition can be bridged using the declarative logic programming paradigm.

In a distributed environment, components implementing identical functionalities may be provisioned in variations in terms of non-functional properties to accommodate different non-functional requirements. Figure 4.9 is an example of a Financial WS portal (FWP), which provides the three types of quote services: stock, fund, and Exchange-Traded Funds (ETF). Those quote services are leased from third-party service providers.

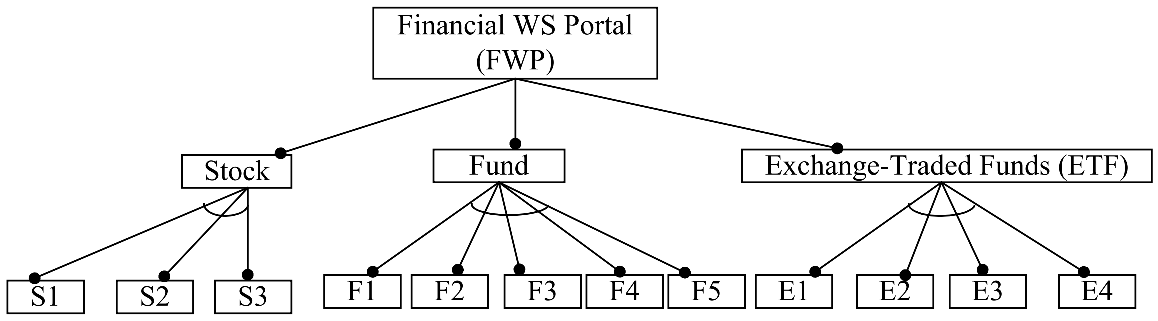


Figure 4.9: Financial WS portal

Every type of service has multiple service providers to choose from, each with a different non-functional properties in terms of QoS (here end-to-end delay) and economical (here service lease charge) properties. The goal of the financial WS portal is to dynamically compose existing third-party services within a certain budget but with shortest end-to-end delay.

Figure 4.9 uses the feature model representation as described in Section 2.4.1 for illustrating the containment relationship of WS. Specifically, the FWP is composed of a Stock quote WS, a Fund quote WS, and an ETF quote WS. Thus each possible FWP corresponds to a *composition tuple* of (*Stock, Fund, ETF*), each item referring to a constituent WS. Each WS has a number of service providers with different end-to-end delays and service charges, which are listed in Table 4.2. It can only choose one of them. The overall non-functional properties for the FWP is calculated as follows (E2ED stands for End-to-End Delay, SC stands for Service Charge):

$$E2ED_{overall} = E2ED_{stock} + E2ED_{fund} + E2ED_{etf}$$

$$SC_{overall} = SC_{stock} + SC_{fund} + SC_{etf}$$

Table 4.2:The non-functional properties for a third-party financial WS provider

WS Type		*End-to-End Delay*	*Service Charge*
Stock	S1	10	200
	S2	20	250
	S3	40	100
Fund	F1	30	170
	F2	50	230
	F3	33	320
	F4	28	145
	F5	17	400
ETF	E1	15	400
	E2	35	300
	E3	25	350
	E4	10	500

Furthermore, there are some constraints associated with the choices of the service providers:

- Bundle sale

  Some services provided from the same company have to be purchased in a bundle. Here the following groups of services have to be purchased in a bundle: (S1, E2), (F4, E1)

- Exclusion sale

  Exclusion constraints can be further applied to the service providers such that there are mutually exclusive service providers that cannot be purchased together with one another. Here such groups of mutual exclusion constraints are:
  (S3, F3, E3), (S1, F5)

Intuitively, the solution space of the component family needs to be explored first to derive all possible composition tuples of (Stock, Fund, ETF) after filtering those non-qualified tuples based on the constraints, then to calculate the shortest end-to-end within an upper limit of service charge. However, once the constraints are changed (e.g., with mutual inclusion or exclusion relationship changed), the solution space exploration algorithm needs to be rewritten to accommodate the change, which is not fit for dynamic composition. In the work presented in this dissertation, Jess is used to resolve this problem.

The Jess specification includes the fact specification and rule specification. The facts for the financial WS portal application includes the non-functional properties of each service provider, and the constraints regarding the qualification of a valid composition tuples. The non-functional properties of each WS are represented with an ordered fact in Jess. For example, for the stock quote provider S1, the corresponding fact definition will be:

```
    (Stock S1 10 200)
```

with corresponds to the tuple of (service type, service name, end-to-end delay, service charge). All facts are illustrated in Figure 4.10.

Figure 4.11 is the Jess query expression to query all the qualified composition tuples together with the corresponding overall end-to-end delay and total service charge. Note those prefixed with “?” represents a regular variable, while those prefixed with $? represents a list variable. Line 3 declares the query parameter, which is the budget allocated for service charges. The query is expected to return all possible composition tuples within the budget. Lines 4-5 bind to the fact base for all possible composition tuples without constraints being applied. Line 7 ensures that the query returns those under budget only. Line 8 creates a list made of the tuple of bounded value of (stock, fund, etf).

```
(Stock S1 10 200)
(Stock S2 20 250)
(Stock S3 40 100)
(Fund  F1 30 170)
(Fund  F2 50 230)
(Fund  F3 33 320)        non-functional properties
(Fund  F4 28 145)
(Fund  F5 17 400)
(ETF   E1 15 400)
(ETF   E2 35 300)
(ETF   E3 25 350)
(ETF   E4 10 500)

(inclusion S1 E2)
(inclusion F4 E1)        constraints
(exclusion S3 F3 E3)
(exclusion S1 F5)
```

Figure 4.10: Fact specification in Jess

```
(defquery search
"Find the shortest end-to-end delay of a composition tuple"
(declare (variables ?budget))
(Stock ?stock ?delay1 ?charge1)
(Fund  ?fund  ?delay2 ?charge2)
(ETF   ?etf   ?delay3 ?charge3)
(<= (+ ?charge1 ?charge2 ?charge3)?budget)
$?para <- (create$ ?stock ?fund ?etf)
(and (inclusion $?inclusionlist)
    (or (=0 (length$ (intersection$ $?inclusionlist $?para)))
       (subsetp $?inclusionlist $?para) ))
(and (exclusion $?exclusionlist)
   (< (length$ (intersection$ $?inclusionlist $?para)) 2))
?delay <- (+ ?delay1 ?delay2 ?delay3)
```

Figure 4.11:Query into fact base in Jess

Lines 9-13 applies the constraints. Specifically, Lines 9-11 ensure the returned tuple satisfies the inclusion constraints (Bundle Sale), which specifies that either the currently bound value list of (stock, fund, etf) has no intersection with any inclusion facts, or the inclusion list is subsumed in the list of (stock, fund, etf). Lines 12-13 ensure the returned tuple satisfies the exclusion constraints (Exclusion Sale) by specifying that there are no two elements in the list of (stock, fund, etf) that appear in any exclusion list.

The query shown in Figure 4.11 returns a collection of qualified composition tuples, together with the non-functional property values such as total end-to-end delay for the corresponding composition tuple. Further rule specification is needed such that whenever the above query returns non-empty results, the composition tuple with the shortest end-to-end delay needs to be returned, which is illustrated in Figure 4.12

In Figure 4.12, a Jess rule is specified: Line 2 represents the condition, while those below Line 3 represent the actions to fire upon the satisfaction of the condition specified in Line 2. In Line 2 the budget of 800 ($) is fed into the query of “search”, which returns all matching results. Note that to ensure those specifications before “=>” are condition specifications, we use pattern binding “<-“ to assign the search result to the ?result variable rather than using the *bind* function, which is an action not a condition. Lines 4-13 iterate through the result sets to get the composition tuple of minimum end-to-end delay. Lines 15-18 is to specify the Jess actions dealing with WS composition through the Jess-WS bridge which is described in the second case study in Section 4.4.2.2. Here sequential aspect weaving (see table 4.1) is used to compose the three WS providers (stock, fund, ETF) together.

```
(defrule FWP
?result <- (run-query* search 750)
 =>
(bind ?minimum-delay -1)
(while (?result next)
  (bind ?delay (?result getString delay))
  (if (< ?minimum-delay ?delay)
   then
   (bind ?minimum-delay ?delay)
   (bind ?stock (?result getString stock))
   (bind ?fund (?result getString fund))
   (bind ?etf (?result getString etf))
  )
)
(if (> ?minumum-delay 0)
  (bind ?aus (new Jess_WS))
  (?aus  apply "after", "", ?stock, "quote", "..", (str-cat ?fund ".quote"))
  (?aus  apply "after", "", ?fund, "quote", "..", (str-cat ?etf ".quote"))
))
```

Figure 4.12: Jess rule for seamlessly integrating WS searching and dynamic WS composition

Based on Figure 4.9, there are totally 3*5*4=60 possible composition tuples, out of which there are 15 qualified composition tuples after applying mutual inclusion and exclusion constraints. With 800 as the budget, there are 6 composition tuples left, among which the composition tuple with shortest end-to-end delay is (S2, F4, E1); the corresponding end-to-end delay is 63.

As it can be seen from Figure 4.12, the WS selection specification and the WS composition specification are unified under the single logic rule specification, the seamless integration of those two are further enabled under a rule inference engine.

4.5 Performance Evaluation

4.5.1 Three-level optimization

Using the profiler to handle all the events generated from all managed execution in CLR is expensive and will degrade system performance significantly. Therefore, we apply optimization at three levels through configuring the profiler as indicated in (7) in Figure 4.3:

1). As the CLR can be launched from a shell, Internet Explorer, ASP.NET, and other customizable CLR hosts for managed execution, we configure the profiler to skip profiling for all non-ASP.NET modules hosted in CLR, which can be filtered easily based on the name of the module that launches the CLR.

2). We could further trim unnecessary profilings based on class name, or CIL method. This is possible because all managed code is translated to CIL, and the CIL level information can be derived from the corresponding WSDL for the WS; this is also necessary to avoid profiling system classes and methods.

3). We mask all unnecessary events except JIT compilation events, which is needed for handling CIL manipulation.

4.5.2 Test setup

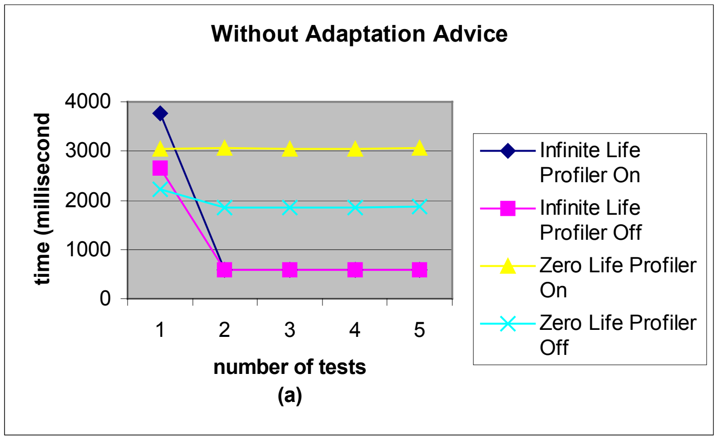
To evaluate the influence of CLR profiling-based WS adaptation on performance, we implemented a simple WS server application with 100 loops for calling a method, which contains only a single addition calculation in its body. We hosted this WS application on a Dell Workstation with Intel XEON CPU 2.2GHx, 1.00GB RAM, which is installed with Win XP professional version 2002 with IIS 5.1, .NET framework version

1.1.4322. We configured the profiler so that the method is to be profiled and adapted with log advice to write to a file a line of strings. A WS stub is generated by compiling the corresponding WSDL for this simple WS application. The WS stub is instrumented together with a simple client application for the client application to call the server-side WS. The client side is hosted on a Dell PC with Intel Pentium 4 CPU 1.80 GHz, 512 MB RAM which resides on the same LAN environment as the server so as to minimize the network influence during the server side performance benchmarking.

Note that the CLR profiling-based approach only applies to managed code to be loaded and JIT compiled. Therefore, we run ASP.NET in the managed mode for profiling WS to realize dynamic adaptation. ASP.NET can load one worker process to handle a pool of WS requests. Once the worker process is launched to serve the first WS requests in the pool, it continues to serve other WS requests in the same pool until the end of its lifecyle without itself being reloaded into CLR, thus it fails to profile the other WS applications in the same pool. Therefore, we adjust the setting for ASP.NET so that a new worker process will be created for each WS request so that each WS call can be captured by the Profiler and thus is adaptable.

4.5.3 Test result evaluation

The goal of our tests is to evaluate how the adjustment of worker process lifetimes (Figure 4.13-a), and the enactment of profiling-based dynamic adaptation (Figure 4.13-b) affect the performance of WS provisioning in the peer-to-peer composition model.



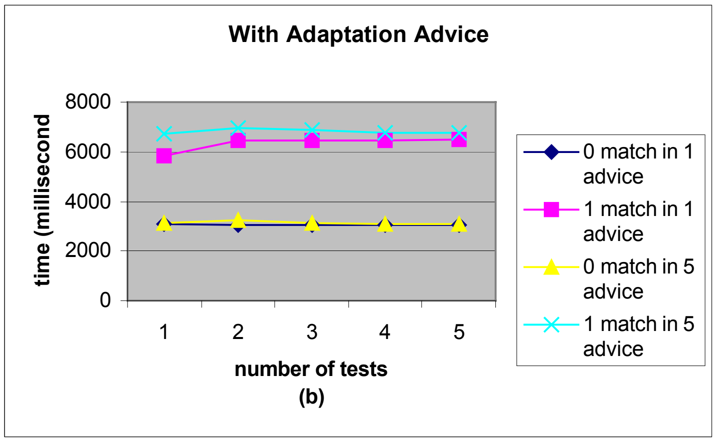


Figure 4.13: Benchmarking dynamic WS adaptation

For the case in Figure 4.13-a, we did not provide any adaptation advice when adjusting the worker process life between *zero life* (a new worker process is created for each WS request) and *infinite life* (the same worker class is used for multiple WS requests). The absence of advice execution will help clarify the influence of the changing life of a worker process on the system performance.

There are significant differences between the first call and the remaining calls for an infinite life case as the first call involves the creation of a new worker class, thus incurring more overhead than the remaining WS calls which reuse the original worker process. Also the presence of profiling does not affect performance much in the case of infinite life, as the worker process is no longer to be reloaded for new WS requests, thus the new WS will not be adapted, and the event handler in the profiling API is ignored. In comparison, the worker process with zero life will incur a performance degradation of 1.7 times slower with profiling on than with profiling off. With the absence of the profiler, the overhead incurred by adjusting from infinite life to zero life will be 3.0 times. With the absence of advice, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class) for this WS provisioning is 3.0*1.7=5.1. Figure 4.14 illustrates the performance degradation.

In Figure 4.13-b, we focus on evaluating the influence of active advice on the overall performance. Therefore, the worker process is set with zero life. We found the number of active advice will not affect the performance linearly, as the AUS are stored in the paging file to be shared by hooks, which constitutes a minor overhead in comparison to that incurred by hook instrumentation and calling of advice. The weaving of a matching advice in the case of zero life in Figure 4.13-b incurs a performance degradation factor of 2.2. Therefore, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class), by synthesizing the result descibed in the preceding paragraph, will be 2.2*5.1=11.2.

In the real world deployment, we can reduce the overhead by setting the worker class to zero life at the adaptation time, then resetting it to infinite time after adaptation is

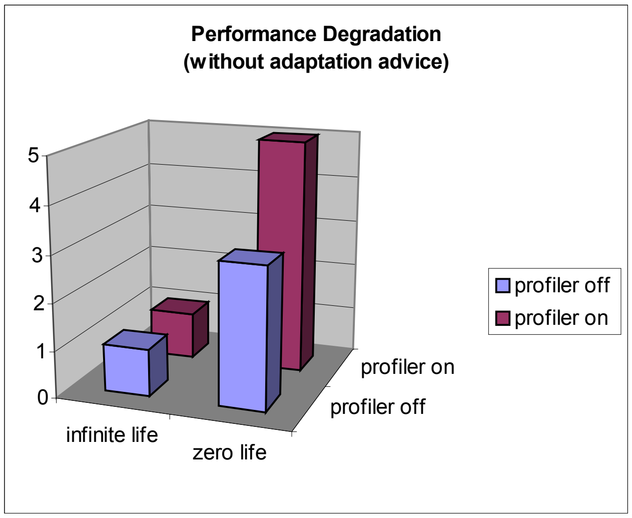


Figure 4.14: Performance degradation with 0 adaptation advice

done. Yet this assumes a predicable adaptation process.

4.6 Related Work

The related work can be classified along several dimensions as follows.

4.6.1 Component composition at different abstraction level and scope

Component composition can be enacted at design level (e.g., [Clarke, 02], [Keller, 98]), and application code level (e.g., [Hölzle, 93], [Mezini, 98], [Seiter, 99]). In contrast, our work on component composition is targeted at the service level, while it is enacted at the intermediate code level without introducing new language constructs. With a lower-level of abstraction, our work enables cross-language component composition, while the above work restricts the component composition to a specific language. Also, none of the aforementioned work on component composition is applied at runtime, which is however necessary in a distributed computing environment.

While the work presented in this dissertation targets WS, the UniFrame project ([Raje, 02], [Olson, 05]) has a more broad vision, which aims at creating a framework for seamless integration of general distributed heterogeneous components. In UniFrame, component composition is also following the peer-to-peer paradigm, which is enabled through discovery services in search of a matching component. Once a searched component does not match the requirement functionally or non-functionally, the search process will be launched again, which exhibits the autonomous features similar to that described in the work presented here. It is expected that the principles of our approach can be integrated into UniFrame as well.

4.6.2 Using AOP for composition and adaptation

The Composition pattern has been proposed in [Clarke, 01], which uses a UML template for specifying composition of crosscutting concerns at a high level and maps sequence diagrams into AspectJ code. Our composition pattern is represented with a comprehensive framework rather than just a design-level pattern. Also a sequence diagram is used here for illustrating the dynamic partnership, with each object in the sequence diagram corresponding to a partner when mapped to dynamic composition specification. In contrast, each object in a sequence diagram is synthesized to an aspect construct in AspectJ in [Clarke, 01]. While AOP has been applied to distributed systems for resolving crosscutting concerns ([Pulvermuller, 99], [Zhang, 03]), here we dedicate AOP to the composition purpose: for composing components handling cross-cutting concerns in a modularized way, as well as for separating composition from components. Moreover, we use the Jess inference engine to autonomously apply aspect weaving for component composition. While the work described in [Yang, 02] also aims at applying an aspect-oriented approach to dynamic adaptation, they only offer a means for making the AOP-based adaptation ready, without presenting any solution on how to use rule engines to trigger the adaptation. Additionally, [Duzan, 04] presents a prototype implementation in the QuO toolkit for an aspect-based approach to programming QoS-adaptive applications. In contrast, our work is targeted at loosely coupled service oriented computing as opposed to tightly coupled distributed object computing in QuO, where adaptation rules are triggered by exceptions thrown from runtime.

4.6.3 Dynamic WS

The work on dynamic WS composition presented in this dissertation complements the existent work on dynamic WS consumption [Verheecke, 04] and dynamic WS orchestration [Charfi, 04]. All three of these apply the AOP principle for modularizing dynamic adaptation. Specifically, in [Verheecke, 04], a Web Services Management Layer (WSML) is introduced using dynamic AOP implementation language JAsCo to enable hot-swapping and runtime management of services. WSML is a Java-based, client-side software layer for consumption of WS. In [Charfi, 04], dynamic service orchestration is realized with AO4BPEL, an aspect-oriented extension to BPEL4WS, which is a static WS composition model. In contrast to those two work, the work presented in this dissertation is based on a P2P paradigm, without designed client/server roles as in [Verheecke, 04], nor centralized composition model as in [Charfi, 04]; the advantage of using P2P model over a centralized composition model has been described in Section 4.3.1.

4.6.4 Handling of non-functional concerns

Our work also incorporates non-functional concerns into WS component composition. Prior work such as IBM's Web Services Level Agreement [Dan, 02] and HP's Web Service Management Language [Sahai, 02] incorporate the notion at a higher-level presentation, rather than address it at a lower-level platform layer. We believe a treatment at a platform layer is necessary for thoroughly addressing non-functional concerns for WS.

4.6.5 Cross-language weaving over .NET

In this work, dynamic WS composition is realized through cross-language weaving at the CIL level in the .NET platform. CLAW [Lam, 02] is also a cross-language aspect weaver in the .NET environment based on the CLR profiling interface. However, in contrast to our instrumentation of hooks into base applications at JIT time, CLAW generates a dynamic proxy at runtime, which lacks the flexibility of adjusting advice weaving decision proactively and retroactively. While CLAW represents an assertive aspect weaving, the work presented in this dissertation further incorporates a rule inference engine for autonomous aspect weaving. Moreover, our work is the first to offer a solution in the middleware context based on CIL code manipulation.

4.7 Summary

This chapter presents a dynamic component composition approach under the service-oriented paradigm in the .NET environment. By using intermediate code manipulation, WS composition is

- possible to cross language boundaries so long as they are CLR-compliant;
- achieved in a non-invasive manner;

Also, WS composition is

- implemented not only in an assertive manner, but also in an autonomous manner using a rule inference engine;
- specified using the AOP paradigm for separating composition specification from components to be composed, and for modularized composition of components handling cross-cutting concerns, with hooks used to weave and unweave advice at runtime proactively and retroactively;
- specified with language neutral XML as the WS components can be exposed with XML-based WSDL. The XML specification is further mapped to a language-neutral type system CTS, with low-level CTS transparent to upper level composition decision makers.

Moreover, the composition rule specification can seamlessly incorporate the WS selection specification based on pattern matching and first order logic of declarative logic programming.

The experimental results show the profiling-based dynamic composition approach is encouraging with the appropriate control over the profiling scope in the WS scenario.

Even though the approach presented in this chapter is .NET based, the principle also applies to other platforms with adequate software vendor support.

CHAPTER 5

FUTURE WORK

This chapter explores some ideas for extending the work presented in the previous two chapters, which are directly related to the model-driven approach, WS modeling, and dynamic component composition.

5.1. Enrich ER-Based Semantic Intermediate Model Operations

In Chapter 3 the ER model is used as a semantic intermediate model for marshaling and unmarshaling models. Just as the compiler can apply code optimization when compiling application code, the marshaling process can be used to apply *optimization* (e.g., reduce redundant models or relationships) for the original modeling language (either UML or domain specific). Among a list of operations yet to be identified, one straightforward operation is the *merge* operation, for which two ER intermediate models can be merged based on their common entities. Consequently, model composition can be realized in a layered manner. Currently, meta-model construction such as in GME is in a monolithic fashion, which reduces human comprehensibility for a large scale meta-model; meta-model construction is error-prone while errors are hard to be localized. A modularized approach for meta-model construction through layered ER intermediate model construction during marshaling and unmarshaling phase is desirable.

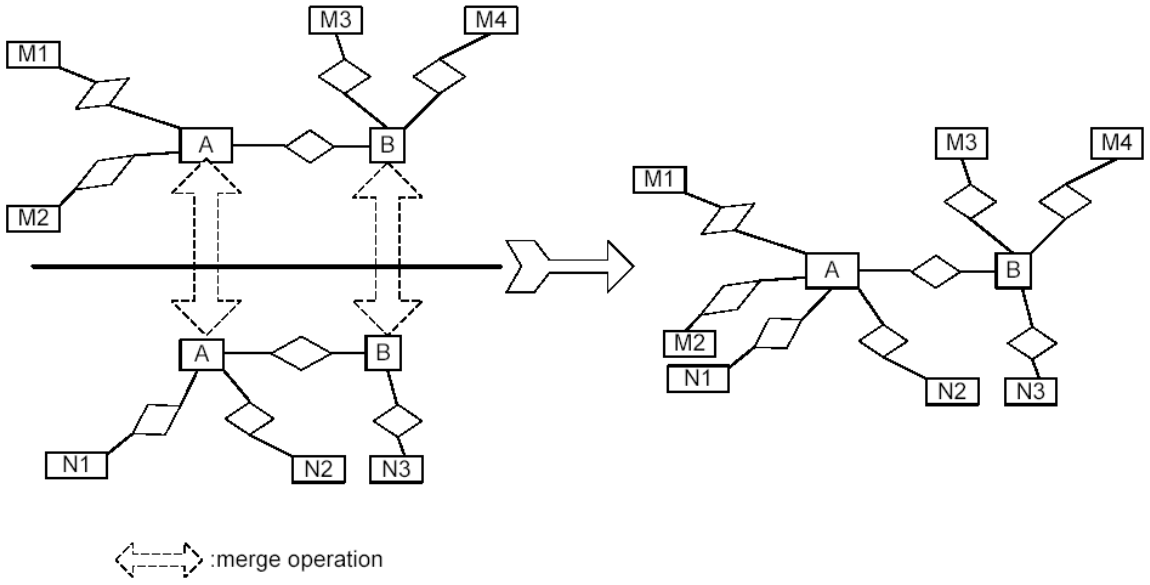


Figure 5.1: Merger operation to enable layered model composition

Nevertheless, it is expected that more such operators are needed to introduce the modularity and abstraction for meta-model /model construction.

5.2. Moving into GME+Eclipse

Tool support is of vital importance in software engineering discipline to promote great ideas. At present, the model marshaling and unmarshaling process described in Chapter 3 lacks tool support to integrate them together and make it a seamless process. In [Zhou, 04], an ER modeling tool has been introduced as a plug-in in Eclipse[44], an open-source extensible Java-based Integrated Development Environment (IDE). With the Java-based BON API available in GME 4, particularly its recent integration into Eclipse itself

[44] http://www.eclipse.org

as a plug-in, Eclipse can be used to seamlessly integrate the model marshaling and unmarshaling process (as shown in Figure 3.2), with the ER modeling plug in exposing one extension point to the GME BON API plug-in within the Eclipse environment.

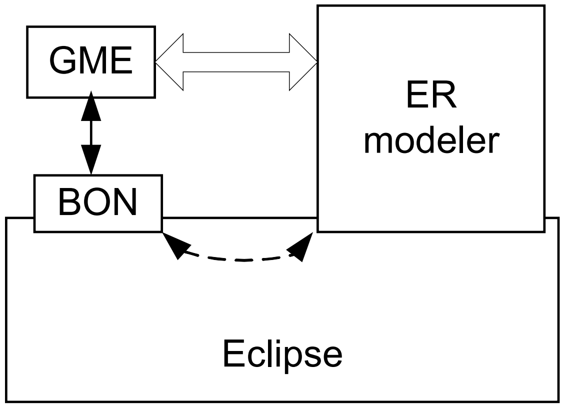


Figure 5.2: Eclipse-based tool integration for seamless model marshaling and unmarshaling

Ultimately, GME meta-models and models can be directly exported to and from the ER modeling environment in the Eclipse environment for seamless model marshaling and unmarshaling. Figure 5.2 provides a tentative architecture for the tool integration described above.

5.3 Aspect Management

The Aspect Library as shown in Figure 4.3 currently is stored in a flat structure, which degrades the searching efficiency as well as restricts the extensibility of the advice library. To facilitate the evolution of the Aspect Library without affecting the dynamic runtime environment, a possible direction could be using the Adaptive Object-Model (AOM) [Yoder, 02] to define an aspect as is shown in Figure 5.3.

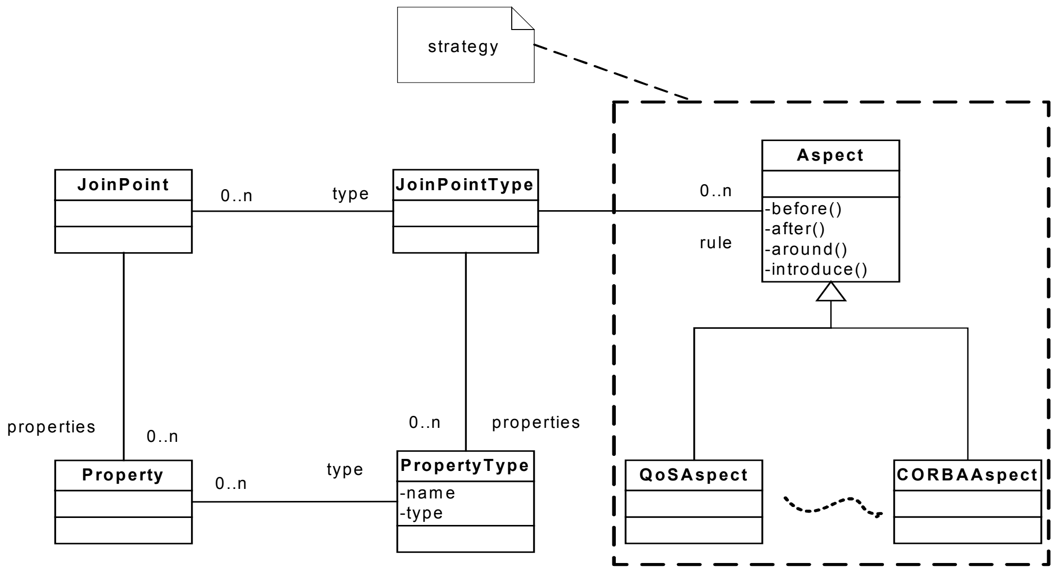


Figure 5.3: Adaptive Object Model for aspect definition

In AOM, the users' object model is interpreted at runtime and can be changed with immediate (but controlled) effect on the system interpreting it. The definition of a domain model and rules for its integrity can be configured by domain experts external to the execution of the program. It is a system that represents classes, attributes, and relationship as metadata. Users change the metadata (object model) to reflect the change in the domain. AOM at its core composes smaller patterns such as TypeObject [Johnson, 98], Property [Foote, 98] pattern for structural description, and Strategy pattern [Gamma, 95] for behavior description. In Figure 5.3, TypeObject and Property patterns are used to represent the join point model, and the Strategy pattern to define the advices to be associated with a service type (a.k.a., the *JoinPointType* in Figure 5.3). As is indicated from

Figure 5.3, one join point type can be associated with multiple aspects, and one join point type also corresponds to multiple join points (represented by the syntactical structure of the base program). As the structure in Figure 5.3 is basically an ER model (with relationship not explicitly modelled, however), this representation can persist in either an object-oriented database or other forms such as an XML file. Consequently, an aspect repository can be comprised of a collection of such representations.

5.4 Rule Management

5.4.1 Model-driven configuration

With Jess as the underlying rule inference engine, rule specification has to be Jess-based, which is not only error prone and takes a new learning curve for the beginners, but also lacks reusability across different rule engine systems, even though the rules are the same semantically. As such, a model-driven approach to raise the rule specification at a higher-level is desirable to address this issue. The MIC paradigm can be leveraged to create a meta-model for rule specifications in general and a Jess interpreter in particular for synthesizing Jess rule specification code from high-level models.

5.4.2 Mobile agent based configuration

As is illustrated in Figure 4.3, the fact base is made public for enabling adaptive composition during run time in the peer-to-peer component composition scenario, in which facts are dynamically added into the base, and upon the matching of a rule, composition strategy can be dynamically applied. In the current implementation updating the fact base uses a *push* mode, i.e., a remote component will write any new facts back to the fact base of the local component, such as in Figure 5.4 (a), remote component B adds new facts into the fact base F1 of a local component A.

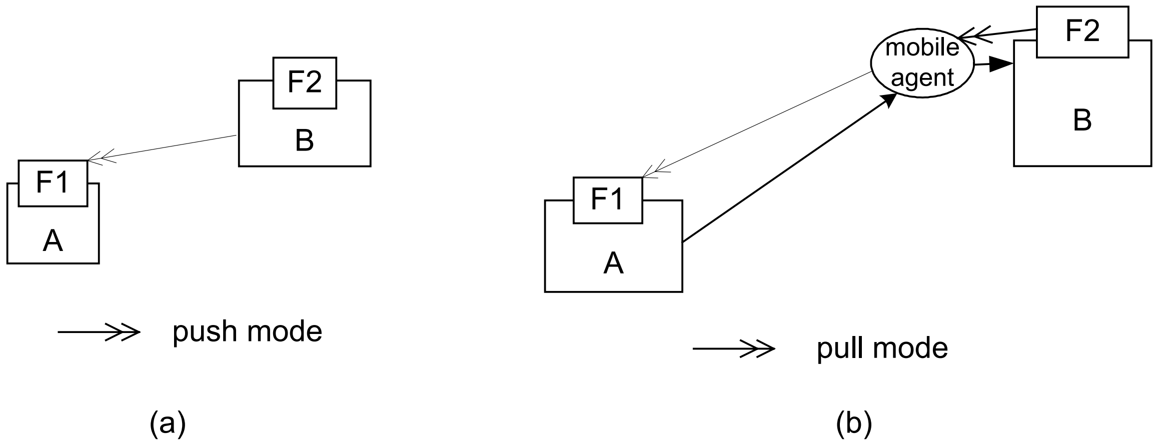


Figure 5.4: Push *vs.* pull mode in updating fact base - A represents a local component; B represents a remote component; F1 and F2 represent the fact base of the corresponding component.

The problem with the pull mode is that there may be some useless fact updates which will waste the bandwidth at the same time. To save the bandwidth in the distributed environment, a mobile agent can be moved around to the destined remote component site to monitor and handle only the interesting facts (which is a *pull* mode) generated at the remote component site only, which in turn can trigger dynamic composition at the local site. Therefore, using the mobile agent, the fact bases can be federated in the P2P component composition scenario as shown in Figure 4.2, and the adaptation based on the remote fact feedback can be more agile and efficient because of the reduced message passing in between. A prototype mobile agent searching environment has been described in [Cao-a, 02] (which is further detailed in Appendix C), with related component specification information on each component site being exposed into the Voyager[45] mobile searching environment for mobile agents to search the hosted components. Likewise, the fact base described in Figure 4.3 can be exposed into a mobile searching environment to be monitored or written by mobile agent. This entails other security issues regarding the fact base access control, which are yet to be investigated as well.

[45] http://www.objectspace.com

CHAPTER 6

CONCLUSION

WS has emerged as a new paradigm of component-based software development, which is based on the open transportation protocol HTTP for interoperation and standard description language XML for service presentation. Moreover, WS brings forth a set of infrastructures following SOA to enable distributed software systems to interoperate across heterogeneous platforms, which expands the scale of software component reuse in the networked environment. With the wide research and development support from both industry and academia, WS is gaining its momentum toward wide adoption in the software industry. As such, two directions toward WS application can be seen in near the future: 1) the migration of legacy distributed software systems toward WS applications; 2) the innovation of new infrastructures, and languages in support of WS application development. The contribution of this dissertation aligns well with those two directions, which is summarized as follows.

For the migration of legacy software system to WS applications, current practice remains on a manual, language specific and ad-hoc process, which is error prone and not efficient. As such, one of the contributions of this dissertation is to present a model-driven approach in Chapter 3 to reengineering legacy software systems to WS applications to resolve the aforementioned issues. This technique is based on the MIC paradigm, but this dissertation in turn contributes to MIC in providing a systematic (as opposed to the existing ad-hoc, error-prone) meta-model construction approach based on the idea of model marshaling and unmarshaling.

Chapter 4 presents an infrastructural contribution to dynamic WS composition. Dynamic WS composition is necessary for both ensuring seamless dynamic service consuming experiences, and adapting services provisioning to meet non-functional requirements such as the QoS and economical concerns while maintaining the service availability. That chapter not only offers a dynamic composition enabling technology based on runtime CIL manipulation on the .NET platform, but also presents the composition paradigm based on the AOP approach which is used not only to separate the composition specification from the component base, but also leverages the AOP for handling the composition of components addressing crosscutting concerns, with hook code instrumented during CIL manipulation time for both retroactive and proactive dynamic adaptation. Moreover, a rule inference engine is introduced into the dynamic composition architecture for both accepting feedback from composing components, and monitoring the current runtime environment for firing composition strategies. This offers a means for autonomous composition complementing the assertive composition. Both autonomous and assertive composition are enacted by the runtime, but for the latter, the composition strategies are specified without considering the changing runtime status, while for the former, they are specified based on the runtime, which is necessary in such scenarios as self-healing and fault-tolerance. Last but not the least, the declarative logic programming, with its pattern binding and first-order logic, offers a sufficient means to specify component selection, which can be seamlessly integrated into component composition rule specification and executed by the rule inference engine.

It can be seen that while WS is a promising technology for evolving distributed component-based software development, it is still in its early stage and both industry and academia research efforts are required to further drive the development of WS technology. As such, a cross-discipline treatment is necessary toward that goal. This dissertation showcases the application of state-of-the-art software engineering techniques as well the programming language approaches to the WS technology development, such as MIC, AOP, logic programming. On the other hand, the approaches presented in this dissertation further contributes to the software engineering field, such as model-marshaling and unmarshaling for model assets interchange and systematic meta-model elicitation, and the dynamic composition architecture, which applies to autonomous distributed component composition other than WS.

LIST OF REFERENCES

[Aßmann, 03] U.Aßmann, Invasive Software Composition, Springer-Verlag, 2003.

[Booch, 99] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[Brand, 01] M.G. J. van den Brand, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, J. Visser, The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. Compiler Construction, In Proc. Int. Conf. On Compiler Construction, April 2001, 365-370.

[Brown, 00] A. W. Brown, Large-Scale Component-Based Development, Prentice Hall, 2000.

[Bryant, 02] B. R. Bryant, B.-S. Lee, Two-Level Grammar as an Object-Oriented Requirements Specification Language,. In Proc. Hawaii Int. Conf. System Sciences, Jan 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/ STDSL01 .pdf.

[Burt, 03] C. C. Burt, B. R. Bryant, R. R. Raje, A. M. Olson, M. Auguston, Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control, In Proc IEEE International Enterprise Distributed Object Computing Conference, September 2003, 159-171.

[Cao-a, 02] F. Cao, B. R. Bryant, R. R. Raje, M. Auguston, A. M. Olson, C. C. Burt, Specifying Heterogeneous Distributed Component, In Proc. Annual ACM Southeast Conference, April 2002, 199-200.

[Cao-b, 02] F. Cao, B. R. Bryant, R. R. Raje, M. Auguston, A. M Olson, C. C. Burt, Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar Using Domain Specific Knowledge, In Proc. Int. Conf. on Formal Engineering Methods, October 2002, 103-107.

[Cao-c, 02] F. Cao, B. R. Bryant, C. C. Burt, R. R. Raje, M. Auguston, A. M. Olson, A Translation Approach to Component Specification, In Proc. ACM Conf. On Object-Oriented Programming, Systems, Languages and Applications Companion, November 2002, 54-55.

[Cao-a, 03] F. Cao, Bryant, Z. Huang, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston, Automating Feature-Oriented Domain Analysis , In Proc. Int. Conf on Software Engineering, Research and Practice, June 2003, 944-949.

[Cao-b, 03] F. Cao, B. R. Bryant, R. R. Raje, M. Auguston, A. M. Olson, C. C. Burt, Assembling Components with Aspect-Oriented Modeling/Specification, In Proc Int. Conf. On Unified Modeling Language Workshop in Software Model Engineering, October 2003, http://www.metamodel.com/wisme-2003/12.pdf.

[Cao-c, 03] F. Cao, B. R. Bryant, C. C. Burt, J. G. Gray, R. R. Raje, A. M. Olson, M. Auguston, Modeling Web Services: toward System Integration in UniFrame, In Proc. World Conf. on Integrated Design and Process Technology, Dec 2003, 83-91.

[Cao, 04] F. Cao, B. R. Bryant, W. Zhao, C. C. Burt, J. G. Gray, R. R. Raje, A. M. Olson, M. Auguston, A Meta-modeling Approach to Web Services, In Proc. IEEE Int. Conf. on Web Services, July 2004, 796-799.

[Cao-a, 05] F. Cao, B. R. Bryant, R. R. Raje, M. Auguston, A. M. Olson, C. C. Burt, A Component Assembly Approach Based on Aspect-Oriented Generative Domain Modeling, Electronic Notes in Theoretical Computer Science, 114, Elsevier Science, 2005, 119-136.

[Cao-b, 05] F Cao, B. R. Bryant, W. Zhao, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston,. Marshaling and Unmarshaling Models using Entity-Relationship Model, In Proc. Annual ACM Symposium on Applied Computing, March 2005, 1553-1557.

[Cao-c, 05] F. Cao, B. R. Bryant, S.-H. Liu, W. Zhao, A Non-Invasive Approach to Dynamic Web Service Provisioning, In Proc. IEEE Int. Conf. on Web Services, July 2005, (*to appear*).

[Cao-d, 05] F. Cao, B. R. Bryant, W. Zhao, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston, Model-Driven Reengineering Legacy Software Systems to Web Services, 2005, (*submitted*) .

[Cao-e, 05] F. Cao, B. R. Bryant, R. R. Raje, A. M. Olson, M. Auguston, W. Zhao, C. C. Burt, A Non-Invasive Approach to Assertive and Autonomous Dynamic Component Composition in Service-Oriented Paradigm, 2005, (*submitted*).

[Charfi, 04] A. Charfi, M. Mezini, Aspect-Oriented Web Service Composition with AO4BPEL, In Proc. European Conference on Web Services 2004, September 2004, 168-182.

[Cheesman, 01] J. Cheesman, J. Daniels, UML Components, Addison-Wesley, 2001.

[Chen, 76] P. P. Chen, The Entity-Relationship Model: Toward a Unified View of Data, ACM Transactions on Database Systems, 1(1), 1976, 9-36.

[Chen, 01] Q. Chen, M. Hsu, Inter-Enterprise Collaborative Business Process Management, In Proc. Int. Conf. on Data Engineering, April 2001, 253-260.
[Choi, 00] J. P.Choi, Aspect-Oriented Programming with Enterprise JavaBeans, In Proc IEEE International Enterprise Distributed Object Computing Conference, September 2000, 252-261.

[Clarke, 01] S. Clarke, R. J. Walker, Composition Patterns: An Approach to Designing Reusable Aspects, In Proc.IEEE Int. Conf. on Software Engineering, May 2001, 5-14.

[Clarke, 02] S. Clarke, Extending Standard UML with Model Composition Semantics, Sci. Comput. Program., 44(1), 2002, 71-100.

[Colan, 04] M. Colan, Service-Oriented Architecture Expands the Vision of Web Services, 2004, http://www-106.ibm.com/developerworks/webservices/library/ws-soaintro.html.

[Czarnecki, 00] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison Wesley, 2000.

[Dan, 02] A. Dan, A. R. Franck, A. Keller, R. King, H. Ludwig, Web Service Level Agreement (WSLA) Language Specification, 2002, http://dwdemos.alphaworks.ibm.com /wstk/comon/wstkdoc/services/utilties/wslaauthoring/WebServiceLevelAgreementLanguage.html.

[Deursen, 02] A. van Deursen, P. Klint, Domain-specific Language Design Requires Feature Descriptions, Journal of Computing and Information Technology, 10(1), 2002, 1-17.

[Devanbu, 96] P. Devanbu, S. Karstu, W. Melo, W. Thomas, Analytical and Empirical Evaluation of Software Reuse Metrics, In Proc. IEEE Int. Conf. on Software Engineering, March 1996, 189-199.

[Duclos, 02] F. Duclos, J. Estublier, P. Morat, Describing and using non functional aspects in component based applications, In Proc Int. Conf. on Aspect-oriented Software Development, April 2002, 65-75.

[Duzan, 04] G. Duzan, J. P. Loyall, R. E. Schantz, R. Shapiro, J. A. Zinky, Building Adaptive Distributed Applications with Middleware and Aspects, In Proc. Int. Conf. on Aspect-Oriented Software Development, March 2004, 66-73.

[Edwards, 04] G. T. Edwards, G. Deng, D. C. Schmidt, A. S. Gokhale, B. Natarajan, Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services, In Proc. Int. Conf. on Generative Programming and Component Engineering, October 2004, 337-360.

[Ernst, 02] J. Ernst, What are the Differences Between a Vocabulary, a Taxonomy, a Thesaurus, an Ontology, and a Meta-Model?, http://www.metamodel.com/article.php?story=20030115211223271.

[Foote, 98] B. Foote, J. W. Yoder, Metadata and Active Object-Models, In Proc. Conference on Patterns Languages of Programs, August, 1998, http://jerry.cs.uiuc.edu/~plop/plop98/ final_submissions/P59.pdf.

[Frankel, 03] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley, 2003.

[Friedman-Hill, 05] E. J. Friedman-Hill, Jess 7.0, The Rule Engine for the Java Platform, Sandia National Laboratories, 2005.

[Gamma, 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Garlan, 00] D. Garlan, R. T. Monroe, D. Wile, Acme: Architectural Description of Component-Based Systems, Foundations of Component-Based Systems, ed. G. T. Leavens, and M. Sitaraman, Cambridge University Press, 2000, 47-68.

[Göbel, 04] S. Göbel, C. Pohl, S. Röttger, S. Zschaler, The COMQUAD Component Model: Enabling Dynamic Selection of Implementations by Weaving Non-Functional Aspects, In Proc. Int. Conf. on Aspect-Oriented Software Development, March 2004, 74-82.

[Gokhale, 04] A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, N. Wang, Model Driven Middleware, Middleware for Communications, ed. Q. Mahmoud, John Wiley and Sons, 2004, 163-187.

[Gough, 02] J. Gough, Compiling for the .NET Common Language Runtime (CLR), Prentice Hall PTR, 2002.

[Graham,02] S. Graham, S. Simeonov, T. Boubez, D. Davis, G. Daniels, Y. Nakamura, R. Neyama, Building Web Services with Java, SAMS, 2002.

[Gray, 01] J. Gray, T. Bapty, S. Neema, J. Tuck, Handling Crosscutting Constraints in Domain-Specific Modeling, Communications of the ACM , 44(10), 2001, 87-93.

[Greenfield, 04] J. Greenfield, K. Short, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004.

[Grundy, 00] J. C. Grundy, Multi-Perspective Specification, Design and Implementation of Components Using Aspects, Int. Journal of Software Engineering and Knowledge Engineering, 10(6), World Scientific, 2000, 713-734.

[Hausmann, 04] J. H. Hausmann, R. Heckel, M. Lohmann, Model-Based Discovery of Web Services. In Proc. IEEE Int. Conf. on Web Services, July 2004, 324-331.

[Heineman, 01] G. T. Heineman, W. T. Councill, Component Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001.

[Higuera, 00] C. de la Higuera, Current Trends in Grammatical Inference, In Proc. Joint IAPR Int. Workshops SSPR & SPR, September 2000, 28-31.

[Hissam, 03] S. A. Hissam, G. A. Moreno, J. A. Stafford, K. C. Wallnau, Enabling Predictable Assembly, Journal of Systems and Software, 65(3), 2003, 185-198.

[Hölzle, 93] U. Hölzle, Integrating Independently-Developed Components in Object-Oriented Languages, In Proc. European Conf. on Object-Oriented Programming, July 1993, 36-56.

[Hunleth, 01] F. Hunleth, R. Cytron, C. Gill, Building Customized Middleware Using Aspect-Oriented Programming, In Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications Workshop on Advanced Separation of Concerns, October 2001.

[ISIS, 01]. ISIS, GME 2000 User's Manual, Version 2.0, Vanderbilt University, 2001.

[Johnson, 98] R. Johnson, B. Wolf, Type Object, In Proc. Conf. on Pattern Languages of Program Design, October 1998, 47-65.

[Kang, 90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Karsai, 03] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-Integrated Development of Embedded Software, IEEE, 91(1), 2003, 145-164.

[Keller, 98] R. K. Keller, R. Schauer, Design Components: Towards Software Composition at the Design Level, In Proc. Int. Conf. on Software Engineering, April 1998, 302-311.

[Kiczales, 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, In Proc. European Conf. on Object-Oriented Programming, June 1997, 220-242.

[Kiczales, 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, In Proc. European Conf. on Object-Oriented Programming, June 2001, 327-353.

[Lam, 02] J. Lam, Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime, Demo at Int. Conf. on Aspect-Oriented Software Development, April, 2002.

[Leavens, 01] G. T. Leavens, M. Sitaraman, Foundations of Component-Based Systems, Cambridge, 2000.

[Lédeczi, 01] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer, 34(11), 2001, 44-51.

[Lédeczi, 03] Á. Lédeczi, J. Davis, S. Neema, A. Agrawal, Modeling Methodology for Integrated Simulation of Embedded Systems, ACM Transactions on Modeling and Computer Simulation, 13(1), 2003, 82-103.

[Lee, 02] B.-S. Lee, B. R. Bryant, Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language, In Proc. Annual ACM Symposium on Applied Computing, March 2002, 932-936.

[Lieberherr, 99] K. Lieberherr, D. Lorenz, M. Mezini, Programming with Aspectual Components, Technical Report, NU-CCS-99-01, 1999, http://www.ccs.neu.edu/research/demeter/papers/aspectual-comps/aspectual.ps.

[Lopes, 03] D. Lopes, S. Hammoudi, Web Service in the Context of MDA, In Proc. Int. Conf. on Web Services, June 2003, 424-427.

[Mantell, 03] K. Mantell, From UML to BPEL: Model Driven Architecture in a Web Services World, http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel/.

[McIlroy,69] D. McIlroy, Mass-produced Software Components, Software Engineering Concepts and Techniques, In Proc. 1968 NATO Conf. on Software Engineering, 1969, 138-155.

[Mezini, 98] M. Mezini, K. J. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development, In Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, October 1998, 97-116.

[Microsoft, 02] Microsoft, Common Language Runtime Profiling, Microsoft Corporation, 2002.

[Newcomer, 02] E. Newcomer, Understanding Web Services, Addison Wesley, 2002.

[Olson, 05] A. M. Olson, R. R. Raje, B. R. Bryant, C. C. Burt, M. Auguston, UniFrame-a Unified Framework for Developing Service-Oriented, Component-Based, Distributed Software Systems, Service-Oriented Software System Engineering: Challenges and Practices, Idea Group, 2005, 68-87.

[Parnas, 72] Parnas, D., On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM*,* 15(12), 1972, 1053-1058.

[Pulvermuller, 99] E. Pulvermuller, H. Klaeren, A. Speck, Aspects in Distributed Environments, In Proc. Int. Symposium on Generative Component-Based Software Engineering, September 1999, 37-48.

[Raje, 00] R. Raje, UMM: Unified Meta-object Model for Open Distributed Systems, In Proc. IEEE Int. Conf. of Algorithms and Architecture for Parallel Processing, 2000, 454-465.

[Raje, 02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components, Concurrency and Computation: Practice and Experience, 14(12), 2002, 1009-1034.

[Sahai, 02] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, F. Casati, Automated SLA Monitoring for Web Services, 2002, http://www.hpl.hp.com/techreports/2002/HPL-2002-191 .pdf

[Seiter, 99] L. M. Seiter, M. Mezini, K. J. Lieberherr, Dynamic Component Gluing, In Proc. Int. Symposium on Generative Programming and Component-Based Software Engineering, September 1999, 134-164.

[Siram, 02] N. N. Siram, R. R. Raje, A. M. Olson, B. R. Bryant, C. C. Burt, M. Auguston, An Architecture for the UniFrame Resource Discovery Service, In Proc.Int. Workshop on Software Engineering and Middleware, May 2002, 22-38.

[Sivashanmugam, 03] K. Sivashanmugam, K. Verma, A. Sheth, J. Miller, Adding Semantics to Web Services Standards, In Proc Int. Conf. on Web Services, June 2003, 395-401

[Straeten, 01] R. V. D. Straeten, J. Brichau, Features and Features Interactions in Software Engineering using Logic, In Proc European Conf on Object-Oriented Programming Workshop on Feature Interaction in Composed Systems, June, 2001.

[Stutz, 03] D. Stutz, T. Neward, G. Shilling, Shared Source CLI – Essentials, O'Reilly Press, 2003.

[Suvée, 03] D. Suvée, W. Vanderperren, V. Jonckers, JAsCo: an Aspect Oriented Approach Tailored for Component-Based Software Development, In Proc. Int. Conf. on Aspect-Oriented Software Development, March, 2003, 21-29.

[Szyperski, 02] C. Szyperski, D. Gruntz, S. Murer, Component Software: Beyond Object-Oriented Programming, 2nd ed., Addison-Wesley/ACM, 2002.

[Ubayashi, 02] N. Ubayashi, T. Tamai, Aspect-Oriented Programming with Model Checking, In Proc. Int. Conf. on Aspect-Oriented Software Development, April, 2002, 148-154.

[Verheecke, 04] B. Verheecke, M. A. Cibrán, W. Vanderperren, D. Suvée, V. Jonckers, AOP for Dynamic Configuration and Management of Web services, Int. Journal on Web Services Research, 1(3), 2004, 25-41.

[Yang, 02] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, P. K. McKinley, An Aspect-Oriented Approach to Dynamic Adaptation, In Proc. The First Workshop on Self-healing Systems, November 2002, 85-92.

[Yoder, 02] J. W. Yoder, R. E. Johnson, The Adaptive Object-Model Architectural Style, In Proc. the Working IEEE/IFIP Conf. on Software Architecture at the World Computer Congress, August 2002, pp. 3-27.

[Zhang, 03] C. Zhang, H.-A. Jacobsen, Refactoring Middleware with Aspects, IEEE Trans. Parallel Distrib. Syst. 14(11), 2003, 1058-1073.

[Zhao, 03] W. Zhao, B. R. Bryant, C. C. Burt , J. G. Gray, R. R. Raje, A. M. Olson, M. Auguston, A Generative and Model Driven Framework for Automated Software Product Generation. In Proc Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, May 2003, http://www.csse.monash.edu.au/~hws/cgi-bin/ CBSE6/ Proceedings/proceedings.cgi.

[Zhou, 04] S. Zhou, C. Xu, H. Wu, J. Zhang, Y. Lin, J. Wang, J. Gray, B. R. Bryant, E-R Modeler: A Database Modeling Toolkit for Eclipse, In Proc. 42th ACM Southeast Conf., April 2004, 160-165.

APPENDIX A

AN EXAMPLE OF GENERATIVE MULTI-STAGE COMPONENT ASSEMBLY

Assume that the component A is a banking domain client component written in Java RMI requesting some banking service from a server. Below is the partial specification of A's CDL:

```
A.0  Component A
A.1  BankOperation:: Service.
A.2  Bank::BusinessDomain.
A.3  Platform::TechDomain.
A.4  requires BankOperations: Platform= "RMI".
A.5  end Component A.
```

Below is an ADL for a QoS measurement aspect stored in the Aspect Library and AUL to use that aspect.

```
 Aspect QoSMeter
  advises: BankOperation.
  before: EventTrace.setBeginTime().
  after: EventTrace.setEndTime().
 end Aspect QoSMeter

apply QoSMeter on A.BankOperation.
```

The above specification of component A weaved with QoSMeter aspects will be translated into the following aspectual component specification:

```
B.0  aspect A
B.1  Bankoperation:: Service.
B.2  Bank::BusinessDomain.
B.3  expect Bankoperations.
B.4  expect wrap Argument. //usage interface
B.5  replace  Bankoperation:   //modification interface
B.6          EventTrace.setBeginTime().
B.7    expected().wrap(<<Platform= "RMI">>).
     //each <<..>> corresponds
     //to each expression in right hand side of ":" of A4
B.8    EventTrace.setEndTime().
B.9  end aspect A
```

B.6 and B.8 are weaved from the QoSMeter aspect. Note those lines prefixed by *expect* denote operations signatures that are expected to be supplied with *advice*, and the *expect-directive* corresponds to the *join points* in AOP. Expected operations are either used (usage interface) or modified (modification interface, preceded with *replace*) in the aspectual component definition. For details please see [Lieberherr, 99]. Also lines B.6-B.8 provide *advice* (reimplementation) for the associated operations to be specified in the *connector* part below.

Assume the component B is a banking domain server component implemented in CORBA providing some banking services.

```
C.0  Component B.
C.1  Withdraw, Deposit:: Port;Bankoperation.
C.2  Bank::Domain.
C.3  Platform::TechDomain .
C.4  provides Bankoperation: Platform= “CORBA”.
C.5  end Component B.
```

Note in line C.1, the two types denoted in the right hand side of “::” means both withdraw and deposit are not only *Ports* (which means they are banking services offered to external components), but also *Bankoperations*,.

Below is an ADL for an Access Control aspect [Burt, 03] from Aspect Library.

```
Aspect AccessControl
  Advises: Service.
  before: Log.Check().
end Aspect AccessControl
```

This aspect can be applied to any *Service* (meta-type, thus applicable to *Withdraw*). Consequently, before each call to Service, Log.Check() will be called to verify the credentials.

The following specification will be translated from the component B specification with the AUL of preceding aspect AccessControl.

```
D.0  connector A-B
D.1  {B.Withdraw, B.Deposit} is  BankOperation.
     //join points
D.2  wrap(Argument):
D.3       apply AccessControl on B.WithDraw, B.Deposit,
D.4          apply RMIAspect on BankOperation when
```

```
                    Argument.getname("Plaform")=="RMI"
D.5  end  connector A-B
```

Note that lines D.2-D.5 further implement the *advice* part for the join points (here, *Withdraw* and *Deposit* operations). The body of *wrap* is to wrap the BankOperation with RMI specific code. This is similar to [Pulvermuller, 99], in which CORBA related operations are modularized as aspects and then woven into the application core to derive a CORBA implementation. The difference here is that, those RMI or CORBA related aspects will be pre-built and retrieved from Aspect Library, and they are represented with high-level specifications (in ADL) rather than at the application code level. Upon weaving in Stage 4 of Figure 2.22, the *wrap* routine in the connector specification will be weaved into the aspectual component specification.

The example illustrated in this section shows that the assembly-related concerns (functional and non-functional) of two components can be handled in separate modules (here in the aspectual component definition and connector specification) from the component specification itself. ADL and AUL provide leverage for the assembly process itself to be easily specified and managed. Consequently the assembly can be implemented by using a weaver to weave assembly-specific *advices* together with component specifications.

APPENDIX B

HOOK INSTRUMENTATION THROUGH BINARY CODE MANIPULATION

This part illustrates the layout of binary code before and after the hook (see Figure 4.4) instrumentation for a random method body as shown below. Below is a C# method that is to be instrumented with hook following the template in Figure 4.4. Please ignore the bad programming style as to using “goto” statement—this is purely used for testing purpose.

```
    .....
    .....
    public static void Main() {

     if(u==10) goto exit;
     u++;
     Console.WriteLine("u: {0}",u);
     long l1=  DateTime.Now.Ticks;
     int n;

     myFoo();
     long l2=  DateTime.Now.Ticks;
     Console.WriteLine("Elapsed: {0}",l2-l1);
     anothermyFoo();
     long l3=  DateTime.Now.Ticks;
     Console.WriteLine("Second time Elapsed (without aspect): {0}",
        l3-l2);

     string g="yes";
     myFffff(g);
     int jp=10;

     if(jp==11)
        goto here;
     else return;

here:
     jp++;
exit:
     return;
   }
     .....
     .....
```

Figure B.1: A C# method to be instrumented with hook

```
0x7E 0x01 0x00 0x00 0x04 0x1F 0x0A 0x33
0x05 0x38 0x9D 0x00 0x00 0x00 0x7E 0x01
0x00 0x00 0x04 0x17 0x58 0x80 0x01 0x00
0x00 0x04 0x72 0x01 0x00 0x00 0x70 0x7E
0x01 0x00 0x00 0x04 0x8C 0x03 0x00 0x00
0x01 0x28 0x02 0x00 0x00 0x0A 0x28 0x03
0x00 0x00 0x0A 0x13 0x06 0x12 0x06 0x28
0x04 0x00 0x00 0x0A 0x0A 0x28 0x02 0x00
0x00 0x06 0x28 0x03 0x00 0x00 0x0A 0x13
0x06 0x12 0x06 0x28 0x04 0x00 0x00 0x0A
0x0C 0x72 0x0F 0x00 0x00 0x70 0x08 0x06
0x59 0x8C 0x06 0x00 0x00 0x01 0x28 0x02
0x00 0x00 0x0A 0x28 0x03 0x00 0x00 0x06
0x28 0x03 0x00 0x00 0x0A 0x13 0x06 0x12
0x06 0x28 0x04 0x00 0x00 0x0A 0x0D 0x72
0x29 0x00 0x00 0x70 0x09 0x08 0x59 0x8C
0x06 0x00 0x00 0x01 0x28 0x02 0x00 0x00
0x0A 0x72 0x7D 0x00 0x00 0x70 0x13 0x04
0x11 0x04 0x28 0x04 0x00 0x00 0x06 0x1F
0x0A 0x13 0x05 0x11 0x05 0x1F 0x0B 0x33
0x02 0x2B 0x02 0x2B 0x08 0x11 0x05 0x17
0x58 0x13 0x05 0x2B 0x00 0x2A
```

(a) Binary code for the C# method before being instrumented with hook.

0x72 0x10 0x01 0x00 0x70 **0x28** 0x07 0x00
0x00 0x0A **0x26** **0x72** 0x86 0x01 0x00 0x70
0x28 0x07 0x00 0x00 0x0A **0x3A** 0xAE 0x00
0x00 0x00 0x7E 0x01 0x00 0x00 0x04 0x1F
0x0A 0x33 0x05 0x38 0x9D 0x00 0x00 0x00
0x7E 0x01 0x00 0x00 0x04 0x17 0x58 0x80
0x01 0x00 0x00 0x04 0x72 0x01 0x00 0x00
0x70 0x7E 0x01 0x00 0x00 0x04 0x8C 0x03
0x00 0x00 0x01 0x28 0x02 0x00 0x00 0x0A
0x28 0x03 0x00 0x00 0x0A 0x13 0x06 0x12
0x06 0x28 0x04 0x00 0x00 0x0A 0x0A 0x28
0x02 0x00 0x00 0x06 0x28 0x03 0x00 0x00
0x0A 0x13 0x06 0x12 0x06 0x28 0x04 0x00
0x00 0x0A 0x0C 0x72 0x0F 0x00 0x00 0x70
0x08 0x06 0x59 0x8C 0x06 0x00 0x00 0x01
0x28 0x02 0x00 0x00 0x0A 0x28 0x03 0x00
0x00 0x06 0x28 0x03 0x00 0x00 0x0A 0x13
0x06 0x12 0x06 0x28 0x04 0x00 0x00 0x0A
0x0D 0x72 0x29 0x00 0x00 0x70 0x09 0x08
0x59 0x8C 0x06 0x00 0x00 0x01 0x28 0x02
0x00 0x00 0x0A 0x72 0x7D 0x00 0x00 0x70
0x13 0x04 0x11 0x04 0x28 0x04 0x00 0x00
0x06 0x1F 0x0A 0x13 0x05 0x11 0x05 0x1F
0x0B 0x33 0x02 0x2B 0x02 0x2B 0x08 0x11
0x05 0x17 0x58 0x13 0x05 0x2B 0x00 0x00
0x72 0x4C 0x01 0x00 0x70 **0x28** 0x07 0x00
0x00 0x0A **0x26** 0x2A

(b) Binary code for the C# method after being instrumented with hook: the underlined part corresponds to the binary code of the original function; the bold numbers represent the opcode for operators in the hook, with the following non-bold numbers the token values for the corresponding operands.

Figure B.2: Hook instrumentation through binary code manipulation

Figure B.2-a illustrates the binary code representation for the method body in Figure B.1, which is the encoding in the operator and token value for the operand. Note that the token value has a pre-defined fixed length, which provides each operator in CIL arguments of fixed length as opposed to arguments of varied length. This facilitates the parsing and verification of CIL code. The binary code output in Figure B.20-a is achieved by parsing the function body (the handle of which is retrieved through CLR profiling interface API) and traversing the (operator, operand) pairs based on length information of the operator and its associated operand. The CLR profiling API used to intercept the JIT event in CLR is:

```
HRESULT JITCompilationStarted(FunctionID functionId,
BOOL fIsSafeToBlock)
```

The `functionID` is the handle for the function being JIT-compiled, which can be used to get its binary representation, and all its metadata information. After the handle to the binary representation of the original method is accessed, the function can be manipulated at the binary code level. Figure B.2-b represents the binary code for the method after being manipulated. Table B.1 shows the corresponding opcode for the operators used in the hook in Figure 4.4.

Table B.1: Opcodes for operators in CIL

operator in CIL	*opcode*
ldstr	0x72
call	0x28
pop	0x26
brtrue	0x3A
ret	0x2A

After the binary code is changed, it can be reset as a function body to be JIT-compiled. This uses the CLR profiling interface API as shown below:

```
HRESULT SetILFunctionBody( ModuleID moduleId,
mdMethodDef method, LPCBYTE pbNewILMethodHeader, ULONG
cbNewMethod )
```

The `moduleId` represents the handle of the given module; the `method` represents the metadata token for captured method; the `pbNewILMethodHeader` is the pointer to the new CIL method header; the `cbNewMethod` is the pointer to the size of the new CIL method header. Note in Figure B.2-b, the ret operator (opcode 0x2A) is removed from the original method end to the new method end to ensure execution of the post-hook.

APPENDIX C

USING MOBILE AGENT FOR COMPONENT SEARCHING

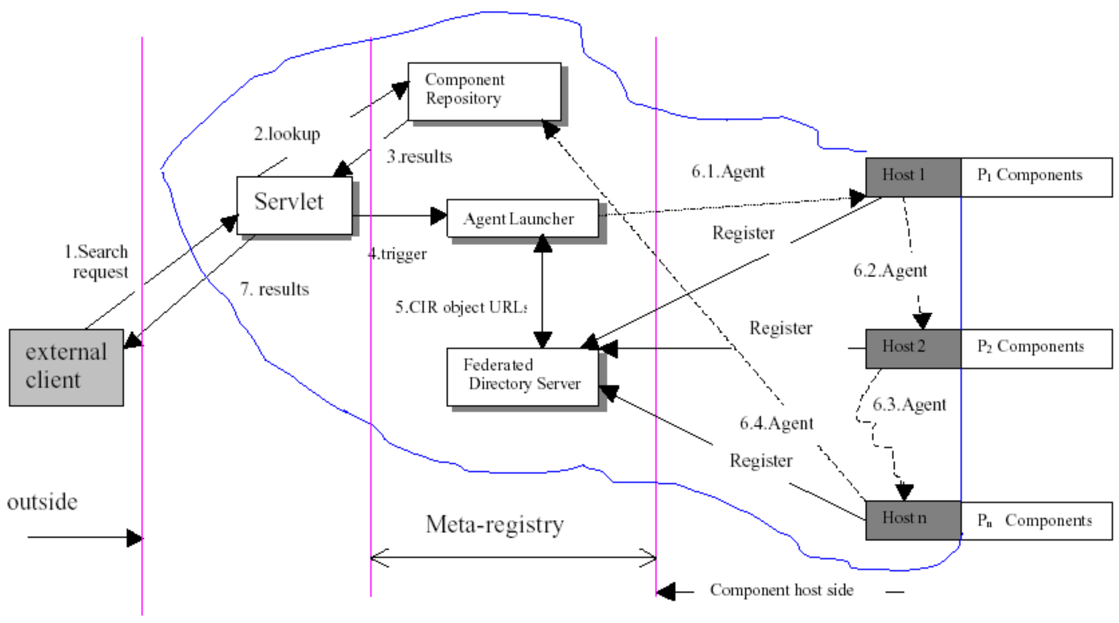


Figure C.1: Architecture of searching component with Voyager Agent; CIR: Component-Info-Retrieval

Figure C.1 illustrates a prototype level example in the Voyager^{TM} environment realizing the mobile agent search of components. Voyager ORB is a high-performance, full-featured object request broker (ORB) that simultaneously supports universal communication between Voyager, CORBA, RMI and DCOM objects. Its innovative dynamic proxy generation removes the need for manual stub generation, and the built-in distributed garbage collection system eliminates the need to explicitly track remote object references. Also remote classloading simplifies deployment and management of application classes. The Voyager ORB also includes a universal, federated and distributed naming service, an activation framework for object persistence, advanced messaging, mobile agent technology and much more.

The searching processes in Figure C.1 are as follows: A client component initiates a search request with its query information regarding service attributes (1). The Servlet parses the request parameters and then looks up the component in the repository (2). The query result is returned to the Servlet (3). If a matching component is already available, the Servlet returns the handle of that component to the client component (7). Otherwise the Servlet launches a search process with the Agent Launcher (4). The Agent Launcher will retrieve the URLs of remote Component-Info-Retrieval (CIR) objects from the Federated Directory Server (5); those URLs are registered leveraging the Voyager ORB's federated distributed naming service by remote objects. The Agent Launcher then sends out mobile agents searching for targeted components through CIR objects (6.1-6.3) and the mobile agents return matching components to the Component Repository for further inquiry by external client components (6.4). Once a matching component is found, client and server components can address to each other directly.

Those parts enclosed by a line are in the Voyager environment. Components do not have to reside in this environment. But their information has to be registered in that environment for a mobile search. Below is the anatomy of the *Meta Registry*

The meta-registry includes the following three elements:

- Component Repository. Component Repository is used to store component information after search is performed for external retrieval at any time. It also has a timestamp attribute indicating its last update time, also a tag showing whether the agent is still in searching status. The outside client can initiates a search and then go ahead with other tasks asynchronously while the search process is underway in the Voyager environment by mobile agents independently

- Federated Directory Server. The federated Directory Server is built on top of the Federated Directory Service of Voyager ORB. Every host in the Voyager environment with components to be searched should register under this Federated Directory Server.This is realized by running a *RegDir* application at the component host side. RegDir integrates the following processes:

  1. Launch voyager server at the host side, export a component-info-retrieval object at some URL. This component-info-retrieval (CIR, in short) object is defined at the meta-registry side, but can be remote loaded without stub, which is one of the strengths of the voyager ORB . Meanwhile, the corresponding URL where this CIR object is exported is registered at the meta-side Federated Directory Server (the headhunter is only passive in this aspect).
  2. The CIR object is only a reference to component information at the host side. Multiple, heterogeneous components may reside at a single host. The CIR object contains a handler to secondary storage (file or database, where component information is stored), which is passed as one of the arguments of "RegDir". In this way, components can achieve autonomy with regard to its actual implementation details.

- Agent Launcher. If the servlet cannot find matching components in the component repository, it will initiate a new search process via the agent launcher. If the agent launcher finds the searching is already underway, it will stop without any further action. Otherwise, a mobile agent is to be created, with the CIR object URLs retrieved as its member variables. Then the mobile agent will move into those URLs one by one, make calls on the CIR object proxy (by looking up the corresponding URLs this CIR object binds to) to retrieve local component information and then proceed to the next URL. If one URL is not accessible, the mobile agent will try the next URL in turn, until all URLs are visited. Then it calls getHome() to get the home URL and move back, duplicating all component information retrieved into the component repository, updating timestamp and some tag for this repository. The agent has a member variable to hold component information during its mobile search.