แบบจำลองเมทาดาตาของซอฟต์แวร์คอมโพเน็นต์ที่นำกลับมาใช้ใหม่ได้ในอินเทอร์เน็ต

นางสมใจ บุญศิริ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2544

ISBN 974-03-1644-1

A METADATA MODEL FOR REUSABLE SOFTWARE COMPONENTS ON THE INTERNET

Mrs. Somjai Boonsiri

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in Computer Engineering

Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University

Academic year 2001
ISBN 974-03-1644-1

Thesis Title	A Metadata Model for Reusable Software Components on the Internet
By	Mrs. Somjai Boonsiri
Field of Study	Computer Engineering
Thesis Advisor	Yunyong Teng-Amnuay, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctor's Degree

---

…….. Dean of Faculty of Engineering
(Professor Somsak Punyakeow, D.Eng.)

THESIS COMMITTEE

…….. Chairman
(Associate Professor Wanchai Rivepiboon, Ph.D.)

……..…… Thesis Advisor
(Yunyong Teng-Amnuay, Ph.D.)

…….. Member
(Associate Professor Somchai Thayarnyong)

…….. Member
(Assistant Professor Supoj Sutanthavibul, Ph.D.)

…….. Member
(Charumatr Pinthong)

นาง สมใจ บุญศิริ : แบบจำลองเมทาดาตาของซอฟต์แวร์คอมโพเน็นต์ที่นำกลับมาใช้ใหม่ได้ในอินเทอร์เน็ต . (A Metadata Model for Reusable Software Components on the Internet) อ. ที่ปรึกษา : อาจารย์ ดร. ยรรยง เต็งอำนวย, จำนวนหน้า 106 หน้า.
ISBN 974-03-1644-1.

การพัฒนาซอฟต์แวร์ในปัจจุบันเป็นการพัฒนาระบบจากการเลือกแล้วรวมซอฟต์แวร์คอมโพเน็นต์เชิงพาณิชย์ (commercial off-the-shelf components) เพื่อให้ทำงานร่วมกัน ดังนั้นคอมโพเน็นต์ที่เลือกมาต้องตรงตามความต้องการ และต้องสามารถทำงานร่วมกับคอมโพเน็นต์อื่นๆได้ด้วย ซึ่งการคัดเลือกและการรวมคอมโพเน็นต์นี้เป็นปัญหาที่ยังไม่มีคำตอบที่แน่ชัด เนื่องจากไม่มีมาตรฐานในการกำหนดคำอธิบายคอมโพเน็นต์ที่ใช้ในการคัดเลือกและการรวมคอมโพเน็นต์เป็นหนึ่งเดียวกัน

งานวิจัยนี้ได้เสนอแบบจำลองเมทาดาตาของซอฟต์แวร์คอมโพเน็นต์ที่นำกลับมาใช้ใหม่ได้เพื่อให้ผู้ค้าคอมโพเน็นต์ใช้ในการอธิบายรายละเอียดเพื่อประกาศคอมโพเน็นต์ และให้ผู้พัฒนาระบบใช้ในการกำหนดคุณสมบัติของคอมโพเน็นต์ที่ต้องการคัดเลือก รวมทั้งสร้างระบบต้นแบบที่ใช้เมทาดาตาซึ่งประกอบด้วยเซตข้อมูลที่อธิบายคุณสมบัติการทำงานร่วมกันของคอมโพเน็นต์ เซตข้อมูลเหล่านี้สามารถใช้ระบุระดับการทำงานร่วมกันของคอมโพเน็นต์ที่นำมารวมกันได้ ด้วยต้นแบบนี้คำอธิบายความต้องการของผู้พัฒนาระบบจะถูกนำไปเปรียบเทียบกับคำอธิบายคอมโพเน็นต์ที่ประกาศไว้ คอมโพเน็นต์ที่มีคุณสมบัติตรงกันจะถูกคัดเลือก แล้วนำมาจัดเป็นกลุ่มที่ผ่านการคัดเลือก กลุ่มเหล่านี้จะถูกจัดคะแนนความเข้ากันได้ระหว่างคอมโพเน็นต์ในกลุ่ม โดยพิจารณาจากกฎการรวมคอมโพเน็นต์ที่ได้กำหนดไว้ ผู้พัฒนาระบบสามารถนำคะแนนเหล่านี้ช่วยในการตัดสินใจเลือกกลุ่มคอมโพเน็นต์ที่เหมาะสมที่สุดได้ เป็นที่คาดหวังว่าเมทาดาตาที่กำหนดจากงานวิจัย และแนวทางการรวมคอมโพเน็นต์นี้จะเป็นจุดเริ่มต้นของการพัฒนาการอธิบายคอมโพเน็นต์ไปสู่ความเป็นมาตรฐานได้ในอนาคต

ภาควิชา วิศวกรรมคอมพิวเตอร์.....
สาขาวิชาวิศวกรรม

ลายมือชื่อนิสิต.....
ลายมือชื่ออาจารย์ที่ปรึกษา.....
ลายมือชื่ออาจารย์ที่ปรึกษาร่วม.....

4171823021 : MAJOR COMPUTER ENGINEERING
KEYWORD : COMPONENT METADATA / COMPONENT INTEGRATION / ENSEMBLE EVALUATION / SOFTWARE REUSABLE COMPONENT / COMPONENT-BASED SOFTWARE DEVELOPMENT

SOMJAI BOONSIRI : A METADATA MODEL FOR REUSABLE SOFTWARE COMPONENTS ON THE INTERNET. THESIS ADVISOR : YUNYONG TENG-AMNUAY, PH.D., 106 pp. ISBN 974-03-1644-1.

The software development trend by selecting and assembling commercial off-the-shelf (COTS) components is widely accepted. This approach implies that the selected components must not only match application requirements but also work well with each other. These component selection and integration issues are not much addressed in research and commercial world as there is no standardized way to specify component descriptions that will be used for selection and integration of components.

This research proposes a metadata model for reusable software components that can be used by component vendors to describe components to be published and by system integrators to specify search criteria for required components. A prototype of the model has been developed to employ the proposed metadata description which comprises integration-related attributes, all of which can be used to determine the degree of compatibility between integrated components. With this prototype, specifications of required components are compared with specifications of published components, and the results are the candidate ensembles that match requirement specifications. These ensembles are then ranked by compatibility scores according to predefined integration rules. System integrators can make further decision on the ensemble that best suits their applications based on the results from the prototype. It is expected that this metadata model and its integration approach can be a starting point for a more standardized component specification and deployment in the future.

Department Computer Engineering	Student's signature ……
Field of Study Computer Engineering	Advisor's signature……..
Academic year 2001…….	Co-advisor's signature……

ACKNOWLEDGMENTS

I wish to express my grateful thanks to my thesis advisor, Dr. Yunyong Teng-Amnuay for his intellectual advice, and consistent support throughout the course of this research. Special thanks to my thesis committee for their helpful comments and suggestion, Associate Professor Dr. Wanchai Rivepiboon, Associate Professor Somchai Thayarnyong, Assistant Professor Dr. Supoj Sutanthavibul, and Lecturer Charumatr Pinthong. I would also like to extend my thank to my supervisors at the Software Engineering Institute, Mr. Robert C. Seacord, and Mr. David A. Mundie for their great efforts and patience to help me achieve the research goal. I greatly appreciate the excellent suggestions by Assistant Professor Dr. Perapon Sophasathit, and Dr. Twittie Seniwongse who reviewed an earlier draft of this research. I am thankful to Mrs. Achara Kuwinpant, Mr. Thiengtrong Vangpatnakuljai, and Miss. Titayarat Intawong for their great help in English writing.

I am very grateful for the assistance from the staff of Department of Mathematics, faculty of science, Chulalongkorn University, Center of Academic Resources (CAR), the SEI, and those I don't mention their names here for their supports and encouragement. I also thank to the ministry of university affairs for financial supporting.

I deeply want to thank Mr. Teera Boonsiri for his love, patience and understanding, financial and technical supports throughout my graduate study. Finally, I wish to thank my parents for their endless love and supporting.

S. Boonsiri
April, 2002.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

PROBLEM STATEMENT

The current software development trend, component-based software development, is introduced comparing to traditional software development. However, the challenging problems are raised and problem solving is proposed including its detail.

1. Motivation

Software requirements of enterprise, complex, and distributed software systems cause software development industry to change from traditional software development approach to component-based software development approach. This approach is based on developing software systems from existing software components or commercial-off-the-shelf (COTS) components, and it is widely adopted in software engineering community. It can reduce cost and time-to-market for software and make reuse more efficient.

The former focuses on building software systems from scratch, whereas the latter focuses on building new systems by selecting and assembling a set of COTS components for an appropriate software architecture.

Traditional software development requires that application developers or system integrators possess medium to high level computer capabilities and specific application experiences. Such demanding prerequisites entail the acquisition of capable developers. Component-based approach, on the contrary, simplifies somewhat the task of built complex software to a certain extent, thus lessening the requirements of competent developers. Nevertheless, the developers are required to have the detailed information of each software component which is quite impossible since software components in the market are “black box” components.

Component-based software development approach by reuse of software components has a great potential for 1) significantly reducing the cost and time to market of large-scale and complex software systems, 2) improving system

maintainability and flexibility by replacing new components to the old ones, 3) enhancing software reliability as components have been undergone evaluation during each use, and 4) enhancing system quality by allowing components and systems to be developed by those who are specialized in the application area and component-based software development (Penix, 2000).

Nevertheless, there are many challenging problems in component-based software development. Two of the major problems are identifying the appropriate components for integration and combining components. This can be accomplished by means of a metadata model for reusable software components defined to effectively identify the right components for system integration.

A metadata means data which describes other data and it is used in many fields such as document cataloguing, database structure, etc. Metadata is the stuff of card catalogues, television guides, taxonomies, tables of contents, so it is quite simply, data about data (Dornfest, 2001).

Therefore, to be successful in component-based software development, software component metadata model and description as shown in Figure 1-1 must be defined and used by component vendors. As depicted in Figure 1-1, metadata model shows the composition of the metadata. Metadata description describes all aspects of metadata, i.e. what attributes, fields, and details of each attribute that represent the components. The component broker collects component information from all component developers and lists this information in metadata form. The system integrator searches the required components by specifying component features in the same format and retrieves metadata of the required components for further execution in building application software systems.

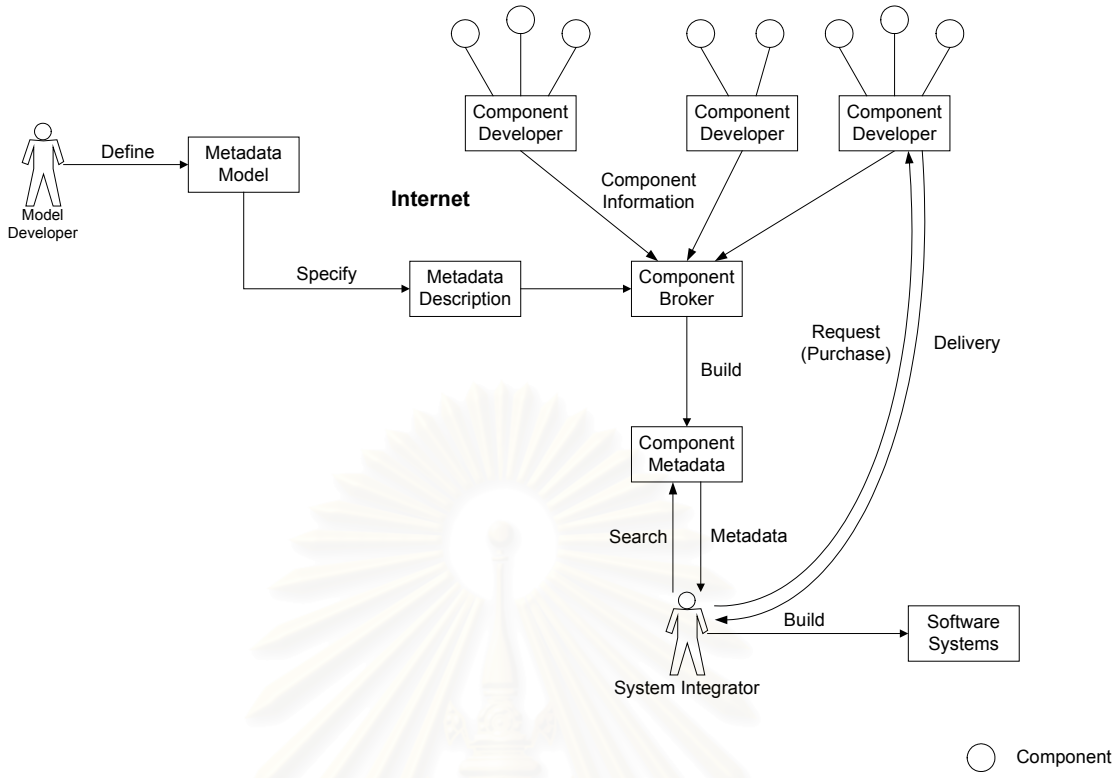

Figure 1-1. A Life Cycle of Software Component Model

2. Research Objectives

Based on the above model, the following tasks are proposed to attain efficient application of software component:

2.1 Define a reusable software component metadata specified in component integration for enterprise component-based software development approach,

2.2 Construct a software component metadata model as defined, and

2.3 Conduct an experiment on the model in component ensemble evaluation prototype.

3. Research Scope

3.1 Survey software component vendors on the Internet,

3.2 Specialize component definition for this research,

3.3 Define a specification for software component metadata,

3.4 Limit software component to component-based software development setting,

3.5 Design a software component metadata model adopts the metadata, and

3.6 Construct a metadata model as defined.

4. Research Methodologies

4.1 Literature survey on research trend.

4.2 Study component-based software development and engineering.

4.3 Study component technologies, methodologies and techniques.

4.4 Study component characteristics and how to define metadata from various component vendors on the Internet.

4.5 In-depth study* on the followings:

4.5.1 Study domain analysis to specific area,

4.5.2 Study component integration and related topics,

4.5.3 Extract related information from various organizations,

4.5.4 Generalize information to define software component metadata,

4.5.5 Experiment using defined metadata for component ensemble evaluation,

4.5.6 Paper writing and submission.

4.6 Plan for future work.

4.7 Documentation

5. Benefits of the Research

The component metadata model can be applied in many software engineering and development activities, particularly in the area of COTS components for Internet applications. Consequently, the benefits for this model are as follows:

5.1 The specification forwarded from this research can be used as a basis in forming the standard for component metadata found on the Internet. This can speed up the deployment and availability of COTS components.

---

* Research as a visiting scientist of the COTS-Based Systems (CBS) initiative at Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, U.S.A. for 1 year (May 2000 – April 2001).

5.2 This model can support software component distribution on the Internet. The component vendors and users specify components' characteristics under common understanding, so required components can be retrieved and applied more than before.

5.3 The prototype developed in this research can be thought as an expert system to help system integrators identify the right components with high compatible possibility.

5.4 The metadata model constructed can be used as the fundamental of component integration and system prediction research development. Consideration of components' common attributes to measure component interoperability enables system integrators to understand the overall software systems in setting up the environment properly.

This dissertation is outlined as follows: Chapter 2 presents concepts of relevant technologies; Chapter 3 elucidates some related work; Chapter 4 describes component metadata model design and prototype architectural design; Chapter 5 explains all implementation of the prototype and the result as well; Chapter 6 discusses a conclusion and future work.

CHAPTER II

RELEVANT TECHNOLOGIES

Many technologies are developed rapidly to improve software engineering methodologies and tools. This research is also based on some of those technologies. In order to have common understanding on basis, component and related topics are described. Furthermore, relevant technologies are explained in brief in this chapter.

1. Component Background

Instead of building software systems from scratch, pre-built or existing software components are used in building new software systems in current market.

1.1 What is a component?

The answer to this question depends on whether selecting on a narrow definition or a broad definition. The narrow and wide views lead to a different appreciation of the relationship between component technology and object-oriented development. Some of the component definitions are as follows:

CBS team at the SEI (Bachman et al., 2000) defines a component as (1) an opaque implementation of functionality, (2) subject to third-party composition, and (3) conformant with a component model.

D'Souza and Wills (D'Souza, 1999) define a software component as "A coherent package of software implementation that (1) can be independently developed and delivered, (2) has explicit and well-specified interfaces for the services it provides, (3) has explicit and well-specified interfaces for the services it expects from others, and (4) can be composed with other components, perhaps customising some of their properties, without modifying the components themselves."

Szyperski (Szyperski, 1997) defines a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Heineman and Councill (Heineman, 2001) define a software component as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

According to current UML specification, UML 1.3 defines a component as a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files (Kobryn, 2000).

A component definition provided by CBDi forum is an identifiable piece of software that describes and/or delivers a set of meaningful services that are only used via well-defined interfaces (Lamela, 2000).

In this research, a component is defined to have a common understanding as code implementation as an independent unit which is not a complete system but can be deployed or assembled with other components. Because enterprise, distributed and complex software application is major consideration in this research so each software component must be only a piece of code or software and composable with other components. Component vendors, platforms, or other business aspects are out of consideration for this research.

1.2 Benefits of components

Over the past decade, many researchers have attempted to improve software development practices by improving design techniques, developing more expressive notations for capturing a system's intended functionality, and encouraging reuse of pre-developed system pieces rather than building from scratch. Component-based development delivers the benefits that have generally eluded low-level object-oriented development. These include (Hurwitz, 1998):

1.2.1 Increased productivity

Support for leveraging prebuilt and existing software assets in the form of components allows developers to focus on highly productive application assembly instead of low-level programming.

1.2.2 Integration of legacy software assets

Encapsulation or wrapping is a technique that allows developers to turn legacy systems into components by hiding the legacy code behind well-defined component interfaces.

1.2.3 Better business focus

Higher levels of abstractions allows developers and business managers to work together to plan, design, and build the application in high-level business terms.

1.2.4 Faster/easier changes

Increased modularity and a lack of dependencies allows developers to modify, add, delete, or swap components quickly as the business needs change.

1.2.5 Investment protection

Through interoperability standards, developers can be assured that standard-based components will work with other components now and in the future.

1.2.6 Ease of use, ease of learning

Through supplier/consumer assembly development models, in which there is a division of labor within IT organizations, developers can quickly become productive in component-based development without extensive retraining.

1.3 Component technologies

Many technologies are presented to solve different problems in each field of computer science. In component-based software development community, there are also many accepted component technologies for solving development problems. In this section, an overview of many popular technologies is provided for preliminary understanding. These includes Microsoft's component technology, Sun Microsystems' JavaBeans, OMG's CORBA, and Web services technology. They are all discussed in sections that follow.

1.3.1 COM, COM+, DCOM, and .NET

Component Object Model (COM) is a general architecture for component software introduced in 1993 (Microsoft, 2000). It provides platform-dependent, based on Windows and Windows

NT, and language-independent component-based applications. COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediate system component. Specially, COM provides a binary standard that components and their clients must follow to ensure dynamic interoperability. This enables on-line software update and cross-language software reuse.

COM+ is the cornerstone of a framework of technologies designed to support the development of large-scale distributed applications on the Windows platform (Heineman, 2001). There are two versions of the framework called Windows Distributed Internet Applications Architecture (DNA) and .NET. COM+ provides run-time services to objects based on their classes' declared needs. COM+ implements its services by intercepting calls between contexts within a single process or across process boundaries. Objects interact with run-time services using object context and call context. COM+ works with both classic COM and the new Common Language Runtime (CLR). Developers can use the COM+ runtime environment as a foundation for building scalable distributed enterprise applications.

Microsoft® .NET is a set of Microsoft software technologies for connecting information, people, systems, and devices. It enables an unprecedented level of software integration through the use of XML Web services: small, discrete, building-block applications that connect to each other—as well as to other, larger applications—via the Internet. .NET connected software delivers what developers need to create XML Web services and stitch them together. The benefit to individuals is seamless, compelling experiences with information sharing (Microsoft, 2002). The .NET Framework is the infrastructure for the overall .NET Platform. The common language runtime and class libraries combine together to provide services and solutions that can be easily integrated within and across a variety of systems. The .NET Framework provides a fully managed, protected, and feature-rich application execution

environment, simplified development and deployment, and seamless integration with a wide variety of languages.

As an extension of the COM, Distributed COM (DCOM), is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. When a client and its component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware the changes of the physical connections.

1.3.2 JavaBeans and Enterprise JavaBeans

Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the Enterprise JavaBeans (EJB) for the server-side component development. The JavaBeans component architecture supports applications of multiple platforms, as well as reusable, client-side and server-side components (Sun, 2000).

Java platform offers an efficient solution to the portability and security problems through the use of portable Java applets. Java provides a universal integration and enabling technology for enterprise application development, including 1) interoperating across multi-vendor servers; 2) propagating transaction and security contexts; 3) servicing mutilingual clients; and 4) supporting ActiveX via DCOM/CORBA bridges.

1.3.3 Common Object Request Broker Architecture (CORBA)

CORBA is an open standard for application interoperability that is defined and supported by the Object Management Group (OMG), an organization of object technology user companies (OMG, 2000). CORBA manages details of component interoperability, and allows applications to communicate with one

another despite of different locations and designers. The interface is the only way that applications or components communicate with each other. Interface Definition Language (IDL) is the most important part of the CORBA standard and the basis for every specification that the OMG adopts. It was first standardized by OMG in 1991. This interface is comparable to component metadata of this research but they are in different formats. IDL is a universally applicable notation for application program interfaces (API). IDL defines an opaque boundary between client code and object implementation (or services) (Mowbray, 1997).

IDL interfaces define the exposed details of distributed objects. Each IDL interface defines a new object type. IDL interfaces can inherit from other interfaces. The complete set of definitions are inherited, and the inherited identifiers cannot be redefined without causing a conflict. The following example defines three interfaces: a common interface for account and more specialized interfaces for checking and savings accounts. The IDL indicates that the checking and savings interfaces inherit all the definitions from the account interface.

```
interface Account {
  // Account definitions
};

interface Checking: Account {
  // Inherits all Account definitions
  // Then adds Checking definitions
};

interface Savings: Account  {
  //  Inherits all Account definitions
  //  Then adds Savings definitions
};
```

IDL supports multiple inheritance; an interface may inherit from several other interfaces. The inherited definitions must not conflict and must be unambiguous.

The major part for interoperability of CORBA system is the Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships between components. Using an ORB, a client can invoke a method on a server object, whose location is completely transparent. The ORB is responsible for intercepting a call and finding an object that can implement the request, pass its parameters, invoke its method, and return the results. The client does not need to know where the object is located, its programming language, its operating system, or any other system aspects that are not related to the interface. In this way, the ORB provides interoperability among applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems as shown in Figure 2-1.

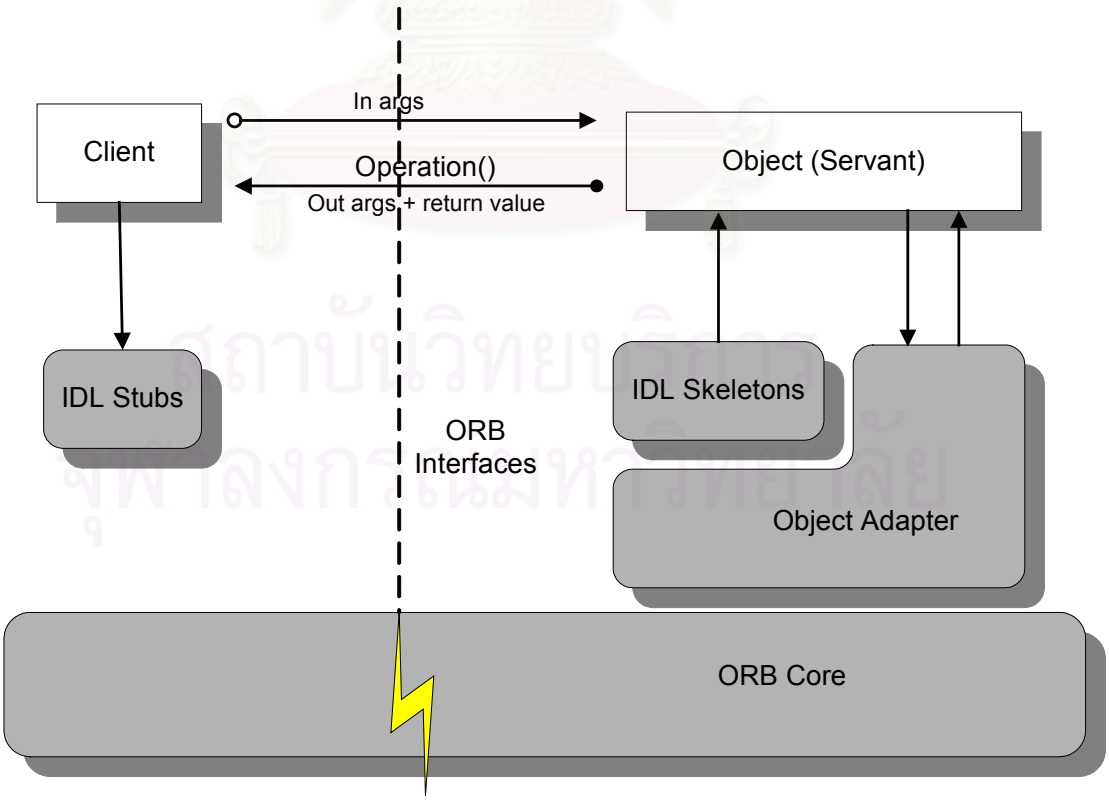

Figure 2-1. CORBA Object Model

Comparison among those component technologies is shown in Table 2-1.

Table 2-1. Comparison of Component Technologies

	COM/DCOM	EJB	CORBA
Development environment	Supported by a wide range of strong development environments	Emerging	Underdeveloped
Binary interfacing standard	A binary standard for component interaction is the heart of COM	Based on COM; Java specific	No binary standards
Compatibility & portability	Not having any concept of source-level standard of standard language binding	Portable by Java language specification; but not very compatible.	Particularly strong in standardizing language bindings; but not so portable
Modification & maintenance	Microsoft IDL for defining component interfaces, need extra modification & maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification & maintenance	CORBA IDL for defining component interfaces, need extra modification & maintenance
Services provided	Recently supplemented by a number of key services	Neither standardized nor implemented	A full set of standardized services; lack of implementations
Platform dependency	Platform dependent	Platform independent	Platform independent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest on the traditional desktop applications	Strongest on general Web clients	Strongest for traditional enterprise computing

1.3.4 Web Services

Web services, an independent application components, are published on to the Web in such a way that other Web application can find and use them. They take the Web to its next stage of evolution, in which software components can discover other

software components and conduct business transactions (Roy, 2001).

Major vendors like IBM, Microsoft, Hewlett-Packard, and Sun, are investing heavily in Web services technology. Web services bring the promise of flexible, open-standards-based, distributed computing to the Internet.

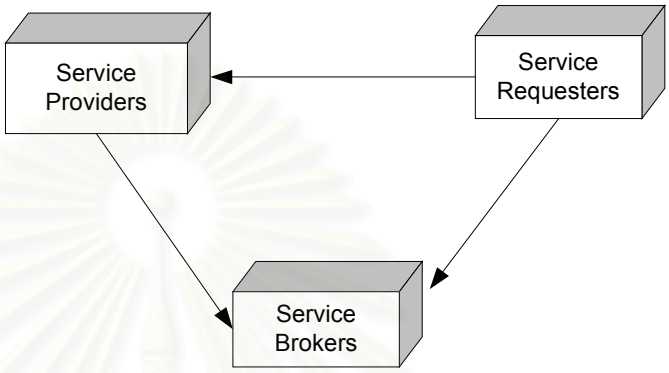

Figure 2-2. Web Services Architecture

Figure 2-2 shows the interaction between service providers, service brokers, and service requesters in the publication, discovery, and consumption of Web services.

Web services are essentially founded upon three major technologies: Web Services Description Language (WSDL); Universal Description, Discovery and Integration (UDDI); and the Simple Object Access Protocol (SOAP).

WSDL is a language which programmers can use to describe the programmatic interfaces of Web services. UDDI lets Web services register their characteristics with a registry so that other applications can look them up. SOAP provides the means for communication between Web services and client applications.

1.4 Component-based software development approach

Modern software systems become more and more large-scale, complex and uneasily controlled, resulting in high development cost, low productivity, unmanageable software quality and high risk to move to new technology. One of the most promising solutions today is the component-based software development (CBSD) approach. This approach is based on the idea that software systems can be developed by selecting appropriate off-the-

shelf components and then assembling them with a well-defined software architecture (Pour, 1998). These COTS components can be developed by different developers using different languages, different platforms and published on the Internet. This can be shown in Figure 2-3, where COTS can be identified or selected from a component repository, and assembled into a new software system.

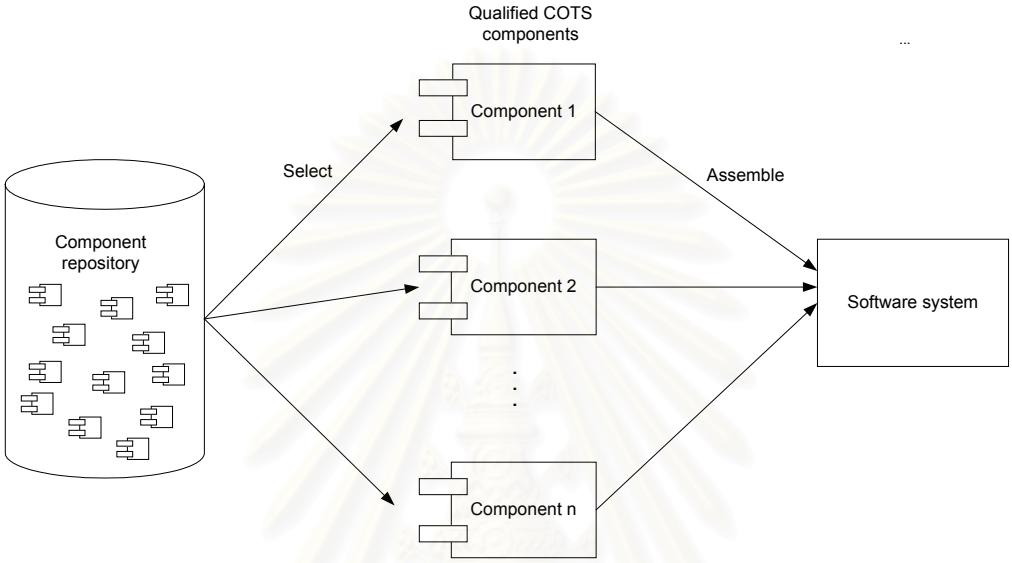

Figure 2-3. Component-Based Software Development

Component-based software development can significantly reduce development cost and time-to-market, and improve maintainability, reliability and overall quality of software systems (Pour, 1999). This approach has raised a tremendous amount of interests both in the research community and in the software industry. Therefore, in the marketplace, more than 99 percent of all executing computer instructions come from COTS products (Basili, 2001).

Component-based software development approach is building new software systems from pre-built components rather than building the systems from scratch, thus the life cycle of component-based software development is different from that of the traditional software development. It can be summarized as follows: (Pour, 1998) (Morisio, 2000) 1) Requirements analysis; 2) Software architecture selection, construction, analysis, and evaluation; 3) COTS component identification and customization; 4) System integration; 5) System testing; 6) Software maintenance.

2. Rule-Based Expert Systems

An expert system is an advanced computer program that can solve difficult problems requiring the use of expertise and experience; it accomplishes this by employing knowledge of the techniques, information, heuristics, and problem-solving process that human expert use to solve such problems (Prerau, 1990).

Rule-based expert systems represent problem-solving knowledge as if…then… rules (Luger, 2002). It is one of the oldest techniques for representing domain knowledge in an expert system. The goal-driven problem solving for analysis of automotive problems is demonstrated below. The example contains four simple rules.

Rule 1: if
the engine is getting gas, and
the engine will turn over,
then
the problem is spark plugs.

Rule 2: if
the engine does not turn over, and
the lights do not come on
then
the problem is battery or cables.

Rule 3: if
the engine does not turn over, and
the lights do come on
then
the problem is the starter motor.

Rule 4: if
there is gas in the fuel tank, and
there is gas in the carburator
then
the engine is getting gas.

There are two premises in rule 1, both of which must be satisfied to prove the conclusion true. In the other word, it can be concluded that the problem is spark plugs if the conditions: the engine is getting gas and the engine will turn over are true.

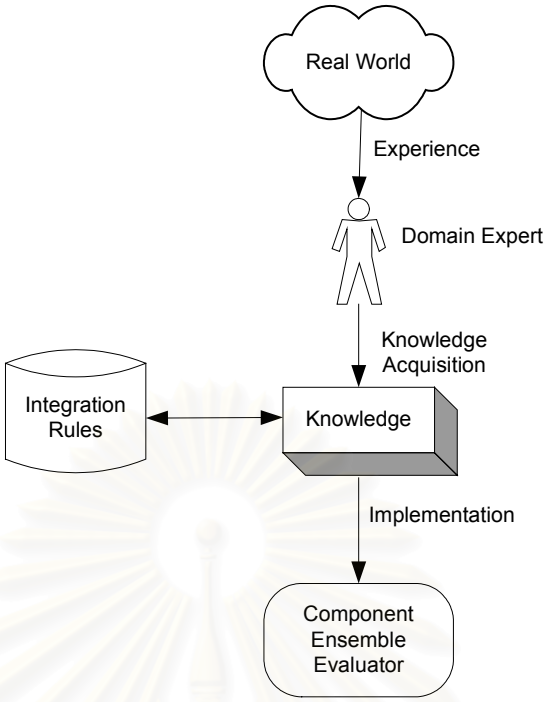

Figure 2-4. A Rule-Based Expert Systems Applied

A domain expert who are experienced component-based software developer in the real world records his knowledge for solving problems in the form of integration rules. Component ensemble evaluator extracts these rules to solve problems in the same conditions.

3. eXtensible Markup Language (XML)

XML is a project of the World Wide Web Consortium (W3C). The development of the XML specification is done under the supervision of W3C's XML Working Group. It is an open specification (non-proprietary) and the current specification (version 1.0) was accepted by the W3C as a Recommendation on Feb 10, 1998. A Recommendation by the W3C indicates that the specification is appropriate for widespread use.

XML is a subset of the Standard Generalized Markup Language (SGML), a complex standard for describing structure and content in documents. It is a meta-language – a language for describing other languages. It is a markup (tag-based) language that is designed to organize data rather than format it. XML looks like HTML, but not exatly the same. It also has start and end tags but instead of defining a bunch of tags, XML allows users to create their own tag pairs and use them in their documents to impart meaning on the data. These tags can be read and used by other's applications as well (Gulbransen, 2000).

XML files are ASCII text, there is no problem incorporating text into XML. XML follows specific rules and is pretty easy to create and parse (read into a program). That makes XML a good choice to use to store internal formats for files that are not usually read by people, but by software instead. Example of BOOK element can be written as follow:

```
<BOOK>
        <ISBN>0-7897-2311-5</ISBN>
        <TITLE> The Complete Idiot's Guide to XML</TITLE>
        <AUTHOR>David Gulbransen</AUTHOR>
        <PUBLISHER>QUE</PUBLISHER>
        <COPYRIGHT>2000</COPYRIGHT>
        <PRICE unit="$">24.99</PRICE>
</BOOK>
```

The BOOK element, called root element, has 6 sub-elements: ISBN, TITLE, AUTHOR, PUBLISHER, COPYRIGHT, and PRICE and their values are shown between start and end tag of each attribute. An attribute can be duplicated if specified.

XML has the following advantages: 1) XML defines the structure of the data and allows XML-based application developers to define their own tags while HTML uses predefined tag, 2) Using XML, document structures can be nested to any level of complexity, 3) any XML document can contain an optional description of its grammar for the use of applications that are required to perform structural validation (Chang, 1998). In addition, the use of XML in enterprise application development reduce complexity and minimizes the coupling between the program and its data. XML has great potential to revolutionize data interchange, presentation, and search on the Internet and Intranet.

Relevant technologies i.e. component background, rule-based expert systems, and XML are explained for fundamental understanding. Related work is discussed in the next chapter.

CHAPTER III

RELATED WORK

There are many principle areas of related work: cataloguing, reusable software library, and component-based software engineering. These topics are described below.

1. Cataloguing

Cataloguing is carried at as part of library documentation for systematic and efficient storage and retrieval. The standard cataloguing methods such as Anglo-American Cataloguing Rules, second edition (AACR2), USMARC are often used. Perhaps one of the most prominent catalogue example is the Internet which is the largest source of electronic information and document. Storing and retrieval formats change from traditional catalogue card to online searching. This leads to the poliferation of software distribution and, hence, shorter software development life cycle. The paradigm of software development alters from building the entire system from scratch to assembling existing components on the Internet. As a consequence, efficient and systematic approaches for storing and retrieval of software components shave many common characteristics with document cataloguing. The latest standard for digital document cataloguing is called Dublin Core, supported by OCLC (Online Computer Library Center), consisting of 15 labeled descriptive elements, namely, title, creator, subject, description, publisher, contributor, date, type, format, identifier, source, language, relation, coverage, and rights. The Dublin Core has the following positive features (Gorman, 1999): simple to learn, repeatable elements, optional elements, extensible to complex applications, embedded invisibly in Web pages, and recognizable by the World Wide Web Consortium. Murphy (Murphy, 1998) described these elements in Table 3-1.

Table 3-1. Elements of Dublin Core Metadata

Element	Description	Examples
COVERAGE	The spatial location and/or duration characteristics of the resource.	North America: 18th century; before 1922.
CREATOR	The person(s) or organizations primarily responsible for the intellectual content of the resource.	Authors; artists; photographers; illustrators.
DATE	The date the resource was made available in its present form.	December 3, 1996 (or 19961203)
DESCRIPTION	A textual description of the content of the resource.	Abstracts for document-like objects or content descriptions for visual resources.
FORMAT	The data representation of the resource.	Text/HTML; ASCII; Postscript file; executable application; JPEG image; MIME type.
LANGUAGE	Language of the intellectual content of the resource.	English; French; Japanese.
OTHER CONTRIBUTORS	Person(s) or organization(s) in addition to those specified in the CREATOR element who have made other significant intellectual contributions to the resource but whose contribution is secondary to the individuals or entities specified in the CREATOR element.	Editors; illustrators; translators; convenors.
PUBLISHER	The entity responsible for making the object available in its current form.	Publisher; university department; corporation.
RELATION	Relationship to other resources.	Chapters in a book; images in a document; items in a collection.
RESOURCE IDENTIFIER	String or number used to uniquely identify the resource.	URL; ISBN; any unique resource file name or key.
RESOURCE TYPE	The category of the resources. [Author's note: The original (1995) description called this the "genre" of the resource.]	Home page; novel; poem; working paper; preprint; technical report; essay; dictionary.
RIGHTS MANAGEMENT	A link to a source that provides information about terms and conditions for use and rights of that use.	Copyright notice; rights-management statement.
SOURCE	The work, either print or electronic, from which this objects is derived, if applicable.	Paper version from which electronic source was transcribed; earlier version of same document.
SUBJECT	The topic of the resource, or keywords or phrases that describe the subject or content of the resource.	Selections from controlled vocabularies such as the Library of Congress Subject Headings.
TITLE	The name of the resource given by the CREATOR or PUBLISHER.	"The Elements of Style"; "Form 1040".

From the example of Dublin core metadata, many of the fields are applicable to the component since a component can be considered as a published material and requires descriptive and functional attributes similar to conventional books and publications. However, these attributes will require much adaptation to fix the nature of the publication in software.

2. Reusable Software Library

Storing, searching, and retrieving software from a repository of reusable components is central to the practice of reuse. Each of these activities relies on the existence of a systematic method of organizing the components so reusers can match existing reusable parts to their current needs (Poulin, 1993).

Poulin (Poulin, 1999) has pointed that large amounts of library metadata that help retrieve components waste time, money, and are difficult to contribute software or retrieve. Reuse libraries should consist of well-designed, domain-specific, and high-quality components.

For software component, there is no good structure or a set of industry standard specifications that provide the necessary information for evaluation of off-the-shelf components (Pour, 1998). In the mean time, the massive amount of software components on the Internet is growing rapidly. Many related research interests have been applied as follows:

Poulin and Werkman (Poulin, 1995) have offered Structured Abstracts (SA), information set required by component users. It consists of 7 items: Computer language and Component type, Domain, Function, Data, Operating System, (Element, …, Element), and Contact. The structure can be written in the form of text as follows: A (Component Language) (Component Type) for (Domain) that provides (Function) on (Data) data running on (Operating System). Principal elements include (Element, …, Element). Contact (Contact) for more information. The entire schema is shown in Figure 3-1.

> C++ classes that provide text buttons and slide bars for the GUI domain. Runs on OS/2 and AIX. Includes documentation, abstract, and test cases. Contact John Smith (smithj@abc.ifs.loral.com)

Figure 3-1. Structured Abstract Sample

According to their experiment with software component using WAIS-indexed database, the result pointed out that the scheme matched user requirements better than keyword search mechanisms.

In this research, component attributes are defined within the realm of component integration so as to allow system integrators making the appropriate decisions on what required components should be integrated to fit well together.

3. Component-Based Software Engineering

Although there are many attempts to define a body of knowledge, code of ethics, accreditation guidelines, and licensing programs but software engineering, while recognizable, is still immature – as evidenced by the significant gap between vision, education, and standard practice (Pour et al., 2000). Many organizations launch these issues Interests of developing component-based software development are described as follows:

Xia et al. (Xia, 2000) developed the Component Integration Model (CIMO), which is a software platform that allows the components written by different software programmer to be integrated and operated into an application without re-compiling. This framework is proprietary structure because a CIMO component is a Microsoft COM object, thus it supports only COM technology. CIMO architecture can be depicted in Figure 3-2 and described in following section.

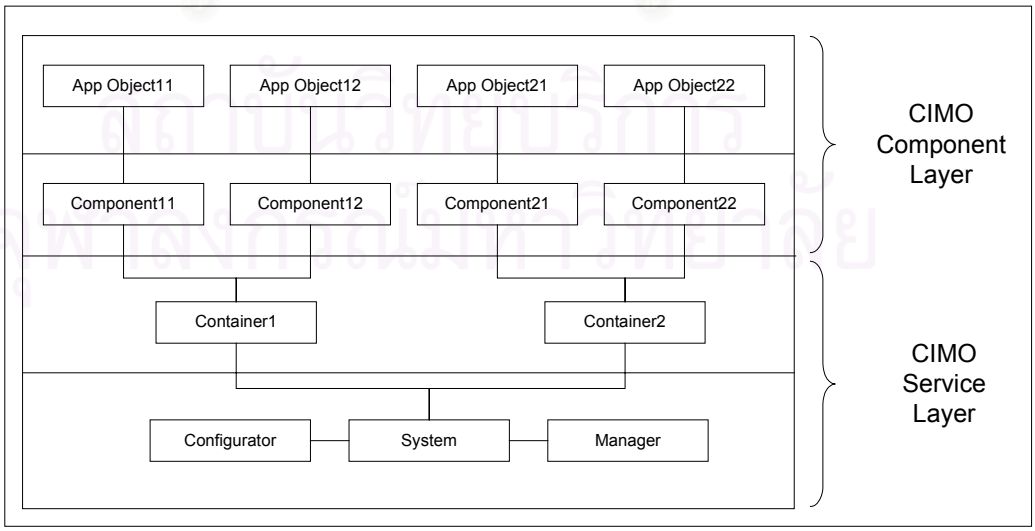

Figure 3-2. CIMO Architecture

CIMO architecture is divided into 2 layers: CIMO Component Layer and CIMO Service Layer. CIMO Component Layer contains CIMO components, which work together and make up a CIMO application. CIMO Service Layer consists of CIMO service components – CIMO Configurator, CIMO Manager, CIMO Containers, and CIMO System. CIMO will provide all the classes in CIMO Service Layer and instructions for application composers to establish the application components. Application composers will write application components and scripts for configuring the application system. Application users will use the final application.

This work differs from this research in that it did not address problems of compatibility among those components.

Yau and Dong (Yau, 2000) presented an approach to use design patterns to automatic generation of the component wrappers for component integration. Their goal is to facilitate CBSD by partially automating component-based software design and implementation. Figure 3-3 described processes of the approach.

Design patterns are organized in a design pattern repository, components and their descriptions can be retrieved from a component repository. According to the user requirements and application specific constraints, components are identified and design patterns are selected to specify component interactions. Software design is generated by instantiating design patterns based on the design pattern instantiation information. If the generated design is consistent with the selected design patterns and satisfies application specific constraints, component wrappers are automatically generated to produce application software. The application software is tested to make sure user requirements and application specific constraints are satisfied.

This work differs from this research in that it generated component wrappers automatically using design patterns in specifying component interactions, while this research only suggests possible component ensemble sets to system integrators.

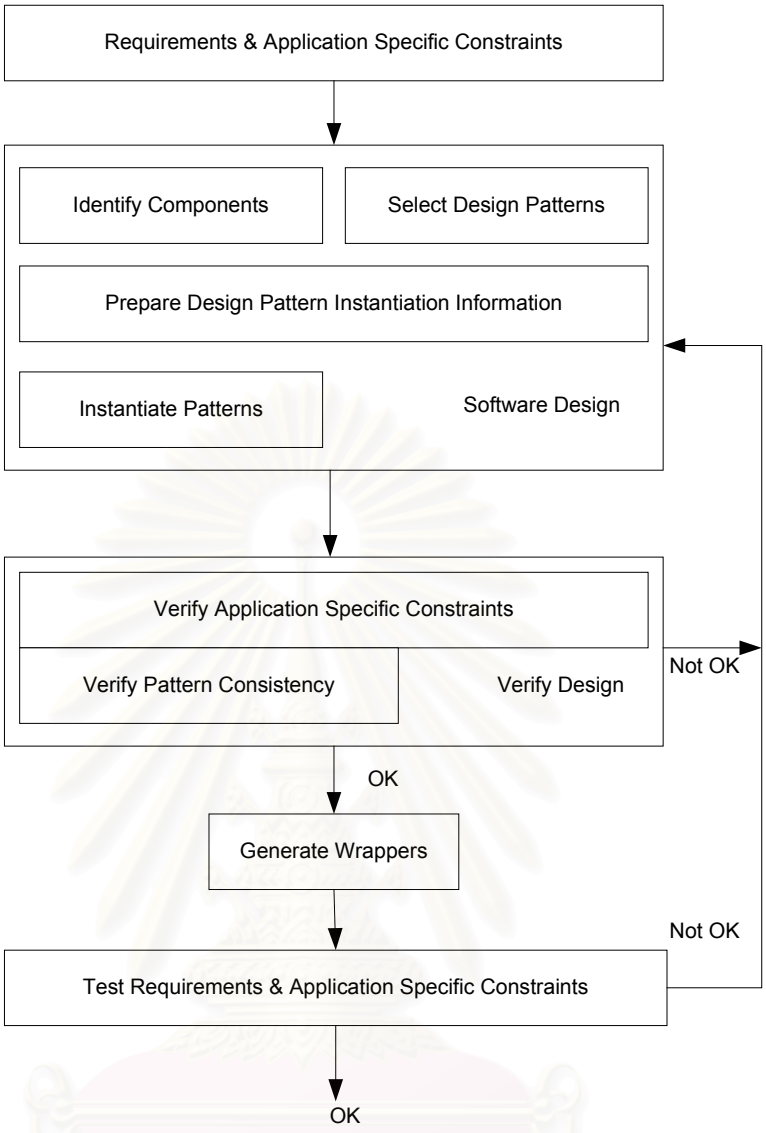

Figure 3-3. CBSD Process Using Design Patterns

To solve the problem of searching the right components, Seacord (Seacord et. al. 1998) developed AGORA, a specialized search engine that automatically generates and indexes a worldwide database of software products. It supports two basic processes: data collection and data search and retrieval. Data collection consists of two sub-processes: location and indexing. Location involves finding components on the Internet. Indexing collects interface information about these components and records it in a local database that serves as a component index. AltaVista is used as a search engine. The overall architecture is depicted in Figure 3-4.

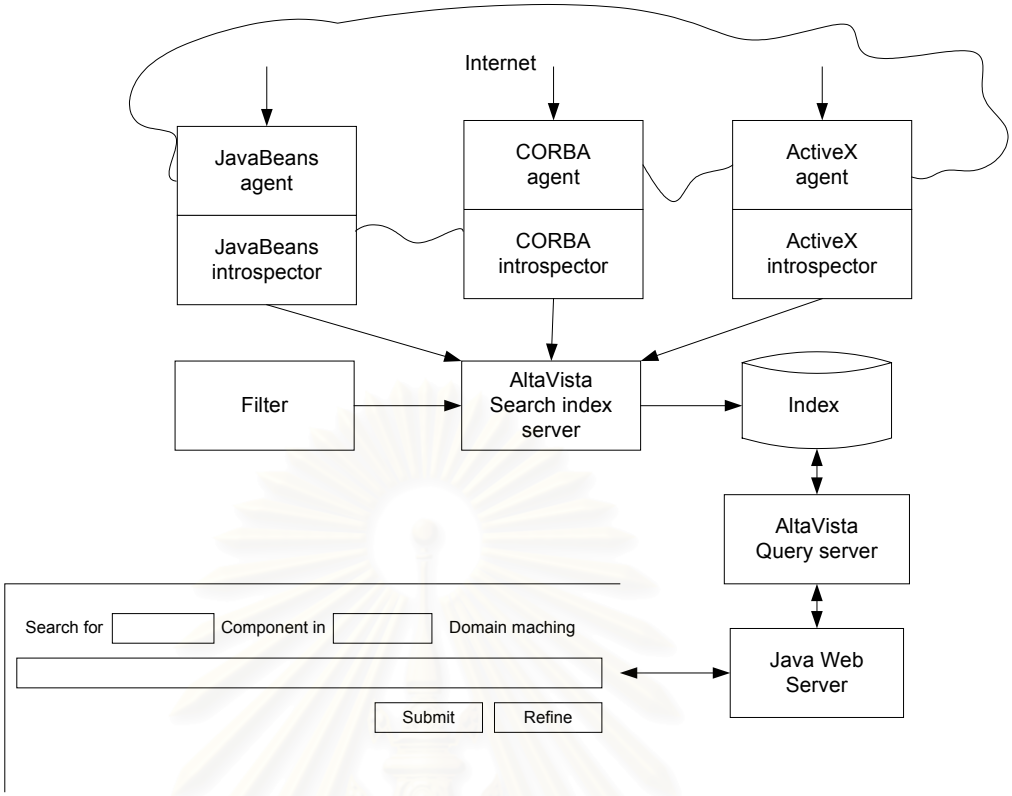

Figure 3-4. AGORA Architecture

Their work directly influences this research in the area of searching software components on the Internet but instead of automatically indexing, the well-defined component metadata approach was focused.

Related work is described to an overview of research status in this field, but none of them is identical to this research. Architectural design of the research is described in the next chapter.

CHAPTER IV

METADATA MODEL DESIGN

To design reusable software component attributes or metadata for component integration, many related researches, relevant technologies, and component vendors on the Internet were surveyed and studied, the result of these studies are used to design the knowledge-based ensemble evaluator of component. This chapter is divided into 2 parts: metadata model design and the prototype architectural design.

1. Metadata Model Design

Two metadata models are designed. The first covers the component metadata for describing component attributes that system integrators can use as criteria in component searching. The other one covers integration-rule metadata for compatibility evaluation between components.

1.1 Component Metadata Design

Prieto-Diaz and Freeman suggested that the characterization of a software component's functionality and its environment suffice for classification (Prieto-Diaz, 1987). The component specification includes information about functionality and environment, and also additional information regarding component characteristics that may affect integration with other components. Gorman stated that the inclusion of additional attributes may improve the effectiveness of component search (Gorman, 1999). It is important to justify the inclusion of these attributes in the component specification and not just add them extemporaneously. Poulin suggests that collecting large amounts of metadata to help retrieve components wastes time and money, and makes the library both difficult to contribute to and difficult to retrieve from (Poulin, 1999). It is therefore necessary to limit the selection of attributes to those that have a significant impact on the suitability of a component for integration. The availability of language bindings, for example, is an important attribute that greatly influences the degree of difficulty involved in integrating a component.

For understanding in notations based on XML document type definition pattern, cardinality operators are described below (Anderson, 2000).

Cardinality operators	Meaning
?	Optional; may or may not appear
*	Zero or more
+	One or more

If no cardinality operator is used, the cardinality is one.

The component metadata or specification has a tree structure as shown in Figure 4-1. The top level or root element i.e. **components** contains one or more element i.e. **component**. For example, a component metadata is composed of 50 components; the root **"components"** contains 50 **"component"** elements. Each **component** element represents software component attributes which contains a component identifier and three main information groups: general information, protocol, and security mechanism information. Each group of information is presented in the form of both tree structure and table in following sections.

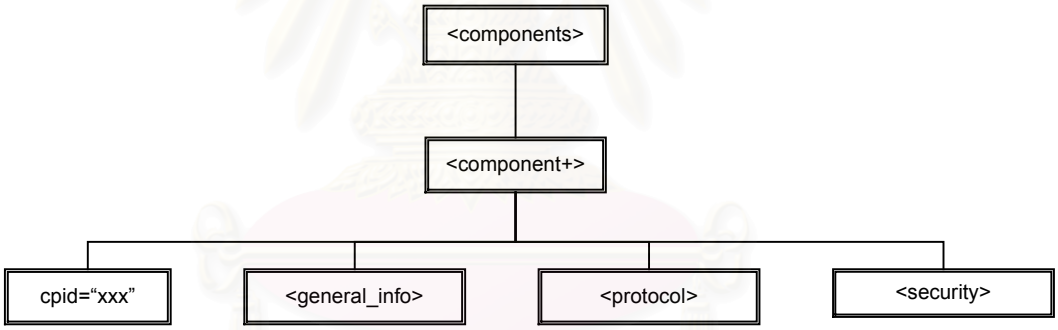

Figure 4-1. Three Structure of Component Metadata

General information is composed of 14 elements: name, version, vendor, platform, function, framework, language, space_req, domain, keywords, gui, cost, license, and lang_support. Vendor element contains 5 sub-elements: name, phone, address, url, and contact. Address element contains 4 sub-elements: street, city, state, and zip. Contact element contains one or more email element. Space_req element contains 2 sub-elements: disk_space and memory_space. Keywords element contains one or more keyword element. Tree representation of general information element can be shown in Figure 4-2.

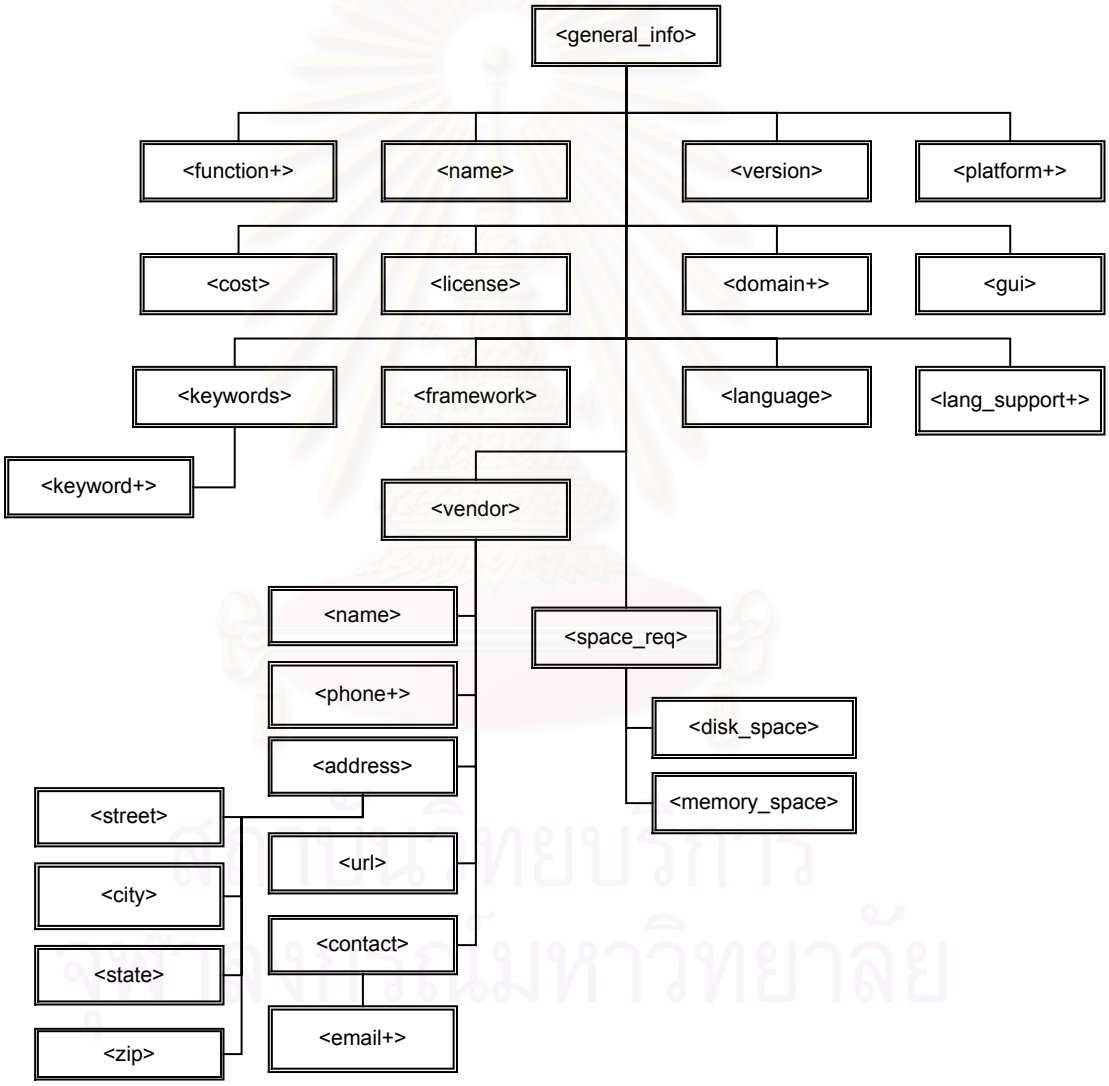

Figure 4-2. Tree Structure of General Information

The attributes of general information can be represented in the form of table as shown below (the plus sign in tree structure is represented in the column **attribute value type** with value "multiple" instead).

Table 4-1. General Information Attributes

Attribute	Description	Data Type	Attribute Value Type	Sample Value
cpid*	component identifier	character	unique	101
general_info				
name*	component name	character	single	Input
version*	component version	character	single	3.5.2
function*	functions of the component	character	multiple	Input
platform*	platform which the component runs on	character	multiple	Windows
vendor				
name*	component vendor name	character	single	ABC
phone	security provider phone number	character	multiple	1 412 2687608
address				
street	provider street address	character	single	1234 Fifth Ave.
city	provider city	character	single	Pittsburgh
state	provider state	character	single	PA
zip	provider zip code	character	single	15267
url	provider's web page address	character	single	www.abc.com
contact				
email	contact person's email addresses	character	multiple	brown@abc.com
framework*	component framework	character	single	CORBA
language*	component implemented language	character	single	Java
space_req				
disk_space	component required disk space	numeric	single	920 KB
memory_space	component required memory space	numeric	single	370 KB
domain*	application domains which the component applied	character	multiple	Financial application
keywords				
keyword	component keyword	character	multiple	report
gui*	Graphical User Interface of the component	character	single	Swing

Table 4-1. General Information Attributes

Attribute	Description	Data Type	Attribute Value Type	Sample Value
cost	cost of the component	numeric	single	$350
license	license term of the component	character	single	One time license
lang_support	languages which the component supports	character	multiple	German, Thai

Note * is a required attribute

From Table 4-1 the significance of various attributes can be expressed as follows.

The ***cpid*** attribute identifies the component in the component repository; it is a unique and required value. Component identifiers can be referenced from other component specifications—allowing a component's interface to be defined in terms of another component.

The ***name*** attribute of general_info element is used to identify and locate the component. The data type is character with single value.

The ***version*** attribute identifies version of the component in the popular form of Major.Minor.Revision such as 1.1.8. The data type of this element is character string with single value.

The ***name*** of ***vendor*** element identifies component vendor name such as Microsoft, Flashline, etc. The data type is character.

The ***street*** element under ***address*** of ***vendor*** element identifies street address of component vendor usually head office. The data type is character string with single value.

The ***city*** element under ***address*** of ***vendor*** element specifies the city that component vendor is located. The data type is character.

The ***state*** element under ***address*** of ***vendor*** element specifies the state that component vendor is located. The data type is character.

The ***zip*** element under ***address*** of ***vendor*** element specifies the zip code that component vendor is located. The data type is character.

This address information is based on address in the United States, it can be modified to appropriate attributes.

The ***contact*** element of ***vendor*** element provides contact information such as sale representatives. The data type of this element is character string. This attribute is borrowed from Poulin's Structured Abstract (Poulin, 1995), item named "Contact".

The ***email*** element under ***contact*** of ***vendor*** element provides email addresses for business contact. The data type of this element is character string.

The ***platform*** attribute specifies what platform which this component runs on. The plus sign means that the component can run on multi-platform, so this element may have one or more platform values. This attribute is borrowed from Poulin's Structured Abstract (Poulin, 1995), item named "Operating System" because it is very important to specify an operating system or platform which the component works on.

The ***function*** attribute defines component functionality. A component may perform more than one function, so more than one value can be defined in this attribute. The data type of this element is character. The functional requirements alone are often specified for system requirements specification. This attribute is also borrowed from Poulin's Structured Abstract, item named "Function" because function attribute is always specified in the system requirement.

The ***framework*** attribute specifies framework which the component is conformant to such as CORBA or EJB. The data type is character.

The ***language*** attribute identifies implemented language of the component such as Java, VB. This attribute is also borrowed from Poulin's Structure Abstract, item named "Computer language" or programming language because it affects compatibility between components.

The ***space_req*** attribute is composed of ***disk_space*** and ***memory_spac***e attributes. They specify minimum disk and memory spaces used by the component respectively. In the other hand, the system integrator may specify disk and memory size of required component in system requirements specification.

The ***domain*** attribute specifies area of application which the component is applied such as financial application domain, manufacturing application domain. Application domain of the component may have more than one domain; therefore, value in this attribute can be repeated. This attribute is also borrowed from Poulin's Structured Abstract, item named "Domain" or environment because it affects the component's performance.

The ***keywords*** attribute is composed of one or more ***keyword*** attribute. The keyword attribute specifies keyword of the component. The system integrator may use keyword as constraints in system requirements specification. The data type is character.

The ***gui*** attribute identifies Graphical User Interface used by the component. The gui attribute value affects compatibility between components. The data type is character.

The ***cost*** attribute identifies cost of the component. It is one of factors that affect system integrator's decision. If this attribute is defined in system requirements specification, it bounds maximum budget of integrator for the component. The data type of this element is numeric.

The ***license*** attribute specifies license term of the component. This element also affects system integrator's decision because it is a part of component's cost. The data type is character string.

The ***lang_support*** attribute identifies languages which this component supports. Some components support more than one language, so that this attribute values can be repeated. The data type of the element is character.

Protocol information is composed of a credential and 4 sub-elements: name, version, provider, and RMI_protocol. Provider element contains 5 sub-elements: name, phone, address, url, and contact. Address element contains 4 sub-elements: street, city, state, and zip. Contact element contains one or more email element. Tree representation of protocol element is shown in Figure 4-3.

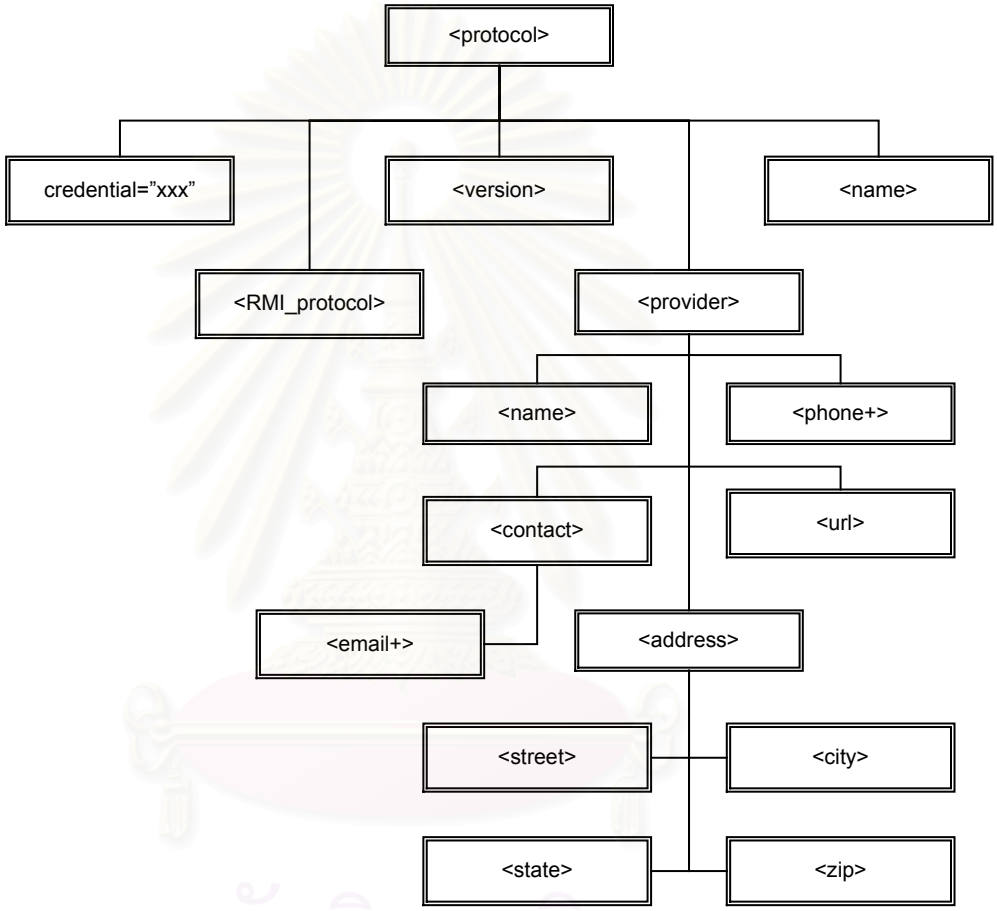

Figure 4-3. Tree Structure of Protocol Information

The elements can be shown in the form of table below.

Table 4-2. Protocol Information Attributes

Attribute	Description	Data Type	Attribute Value Type	Sample Value
protocol				
credential	Information sources	character	single	provider
name*	communication protocol name	character	single	IIOP
version*	communication protocol version	character	single	2.0
provider		character	single	
name	component vendor name	character	single	NetCo
phone	protocol provider phone number	character	multiple	1 624 1680608
address				
street	provider street address	character	single	905 Henrry Street
city	city	character	single	Glendale
state	state	character	single	AZ
zip	zip code	character	single	76823
url	provider's web page address	character	single	www.netco.com
contact				
email	contact person's email addresses	character	multiple	dixon@netco.com
RMI_protocol*	Remote Method Invocation protocol version	character	single	RMI1.2

Note * is a required attribute

From Table 4-2 the significance of various attributes can be expressed as follows.

The ***name*** attribute under ***protocol*** attribute specifies communication protocol name which the component uses. Components can communicate with each other via the protocol. The widely used protocols are IIOP (Internet Inter-ORB Protocol), TCP/IP (Transmission Control Protocol/ Internet Protocol) etc.

The ***version*** attribute under ***protocol*** attribute specifies communication protocol version in the popular form of Major.Minor.Revision such as 3.2.1. The data type of this element is character string with single value.

The ***name*** of ***provider*** attribute under ***protocol*** attribute provides protocol provider name such as Inprise.

The ***street*** element under ***address*** of ***provider*** element identifies street address of component vendor usually head office. The data type is character string with single value.

The ***city*** element under ***address*** of ***provider*** element specifies the city that protocol provider is located. The data type is character.

The ***state*** element under ***address*** of ***provider*** element specifies the state that protocol provider is located. The data type is character.

The ***zip*** element under ***address*** of ***provider*** element specifies the zip code that protocol provider is located. The data type is character.

This address information is based on address in the United States, it can be modified to appropriate attributes.

The ***contact*** element of ***provider*** element provides contact information such as sale representatives. The data type is character string. The data type of this element is character string. This attribute is borrowed from Poulin's Structured Abstract (Poulin, 1995), item named "Contact".

The ***email*** element under ***contact*** of ***provider*** element provides email addresses for business contact. The data type is character string.

The ***RMI_protocol*** attribute identifies Remote Method Invocation protocol version used by the component.

Security element contains zero or more elements of confidentiality, authentication, and nonrepudiation (represented by question mark). Each of which contains a credential and 2 sub-elements: name, and provider. Provider element contains 5 sub-elements same as "vendor" and its child elements. This tree structure of security element can be shown in Figure 4-4. In this research, all elements of security element are optional because integration rule database does not contain compatibility rules on security mechanism.

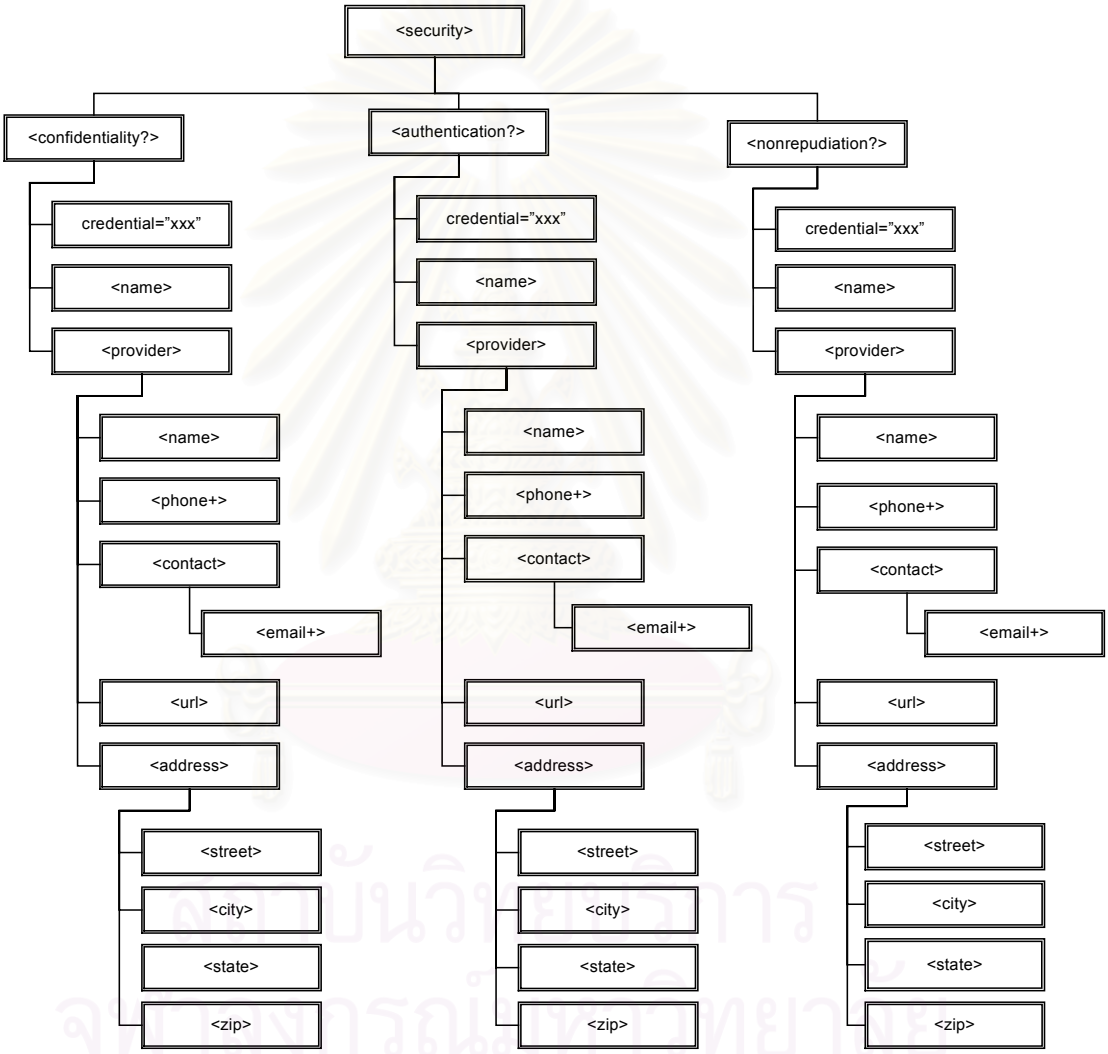

Figure 4-4. Tree Structure of Security Information

The attributes can be shown in the form of table as follow.

Table 4-3. Security Information Attributes

Attribute	Description	Data Type	Attribute Value Type	Sample Value
security				
confidentiality				
credential	Information sources	character	single	testing
name	confidentiality mechanism name	character	single	encryption
provider		character	single	
name	security provider name	character	single	Securicor
phone	security provider phone number	character	multiple	1 810 3485653
address				
street	provider street address	character	single	45 Bayard Street
city	city	character	single	Sanfrancisco
state	state	character	single	CA
zip	zip code	character	single	35678
url	provider's web page address	character	single	www.securicor.com
contact				
email	contact person's email addresses	character	multiple	anderson@secure.com
authentication				
credential	Information sources	character	single	provider
name	authentication mechanism name	character	single	PKI
provider		character	single	
name	security provider name	character	single	Securicor
phone	security provider phone number	character	multiple	1 810 3485653
address				
street	provider street address	character	single	45 Bayard Street
city	city	character	single	Sanfrancisco
state	state	character	single	CA
zip	zip code	character	single	35678
url	provider's web page address	character	single	www.securicor.com
contact				
email	contact person's email addresses	character	multiple	anderson@secure.com
nonrepudiation				
credential	Information sources	character	single	provider

Table 4-3. Security Information Attributes

Attribute	Description	Data Type	Attribute Value Type	Sample Value
name	nonrepudiation mechanism name	character	single	certificates
provider		character	single	
name	security provider name	character	single	Certify
phone	security provider phone number	character	multiple	1 524 8986543
address				
street	provider street address	character	single	89 Melwood Ave.
city	city	character	single	Philadelphai
state	state	character	single	PA
zip	zip code	character	single	22267
url	provider's web page address	character	single	www.certify.net
contact				
email	contact person's email addresses	character	multiple	wallnau@certify.net

From Table 4-3 the detail of various attributes can be expressed as follows.

The ***credential*** element of ***confidentiality*** of ***security*** attribute provides sources of information about this mechanism such as provider, testing.

The ***name*** attribute under ***confidentiality*** of ***security*** attribute specifies name of confidentiality mechanism used by the component. Confidentiality assures that unintended third parties cannot view information sent between two communication parties. Encryption is the most widely used mechanism for providing confidentiality over an insecure medium (Wallnau, 2002).

The ***name*** of ***provider*** attribute under ***security*** attribute provides security provider name such as NetSecure.

The ***street*** element under ***address*** of ***provider*** element identifies street address of security provider usually head office. The data type is character string with single value.

The ***city*** element under ***address*** of ***provider*** element specifies the city that security provider is located. The data type is character.

The ***state*** element under ***address*** of ***provider*** element specifies the state that security provider is located. The data type is character.

The ***zip*** element under ***address*** of ***provider*** element specifies the zip code that security provider is located. The data type is character.

This address information is based on address in the United States, it can be modified to appropriate attributes.

The ***contact*** element of ***provider*** element provides contact information such as sale representatives. The data type of this element is character string. This attribute is borrowed from Poulin's Structured Abstract (Poulin, 1995), item named "Contact".

The ***email*** element under ***contact*** of ***provider*** element provides email addresses for business contact.

The ***credential*** element of ***authentication*** of ***security*** attribute provides sources of information about this mechanism such as provider, testing.

The ***name*** attribute under ***authentication*** of ***security*** attribute specifies name of authentication mechanism used by the component. Authentication always comes with Identification (I&A). This includes how to access private assets such as a computer account, the widely used mechanism is Public Key Infrastructure (PKI) (Wallnau, 2002).

The ***name*** of ***provider*** attribute under ***security*** attribute provides security provider name such NetSecure.

The ***street*** element under ***address*** of ***provider*** element identifies street address of component vendor usually head office. The data type is character string with single value.

The ***city*** element under ***address*** of ***provider*** element specifies the city that component vendor is located. The data type is character.

The ***state*** element under ***address*** of ***provider*** element specifies the state that component vendor is located. The data type is character.

The ***zip*** element under ***address*** of ***provider*** element specifies the zip code that component vendor is located. The data type is character.

This address information is based on address in the United States, it can be modified to appropriate attributes.

The ***contact*** element of ***provider*** element provides contact information such as sale representatives. The data type of this element is character string. This attribute is borrowed from Poulin's Structured Abstract (Poulin, 1995), item named "Contact".

The ***email*** element under ***contact*** of ***provider*** element provides email addresses for contact.

The ***credential*** element of ***nonrepudiation*** of ***security*** attribute provides sources of information about this mechanism such as provider, testing.

The ***name*** attribute under ***nonrepudiation*** of ***security*** attribute specifies name of nonrepudiation mechanism used by the component. Nonrepudiation is the inability to disavow an act. In other words, evidence exists that prevents a person from denying an act. Systems that use mechanism for nonrepudiation are more secure than those that do not. Basis mechanisms for nonrepudiation i.e. public/private key cryptography, digital signatures, and certificates (Wallnau, 2002).

The ***name*** of ***provider*** attribute under ***security*** attribute provides security provider name such NetSecure.

The ***street*** element under ***address*** of ***provider*** element identifies street address of component vendor usually head office. The data type is character string with single value.

The ***city*** element under ***address*** of ***provider*** element specifies the city that component vendor is located. The data type is character.

The ***state*** element under ***address*** of ***provider*** element specifies the state that component vendor is located. The data type is character.

The ***zip*** element under ***address*** of ***provider*** element specifies the zip code that component vendor is located. The data type is character.

This address information is based on address in the United States, it can be modified to appropriate attributes.

The ***contact*** element of ***provider*** element provides contact information such as sale representatives. The data type of this element is character string. This attribute is borrowed from Poulin's Structured Abstract (Poulin, 1995), item named "Contact".

The ***email*** element under ***contact*** of ***provider*** element provides email addresses for contact.

1.2 Integration Rule Metadata Design

Components in each ensemble are evaluated for compatibility based on a repository of software engineering integration rules. The integration rule repository may be extended by system integrators and component vendors. Integration rules typically reflect known compatibilities and incompatibilities between products. The discovery and refinement of these rules is a normal part of the system integration

process. This makes it necessary to provide mechanisms for adding new integration rules to the repository, and to modify and delete existing rules as shown in Figure 4-5.

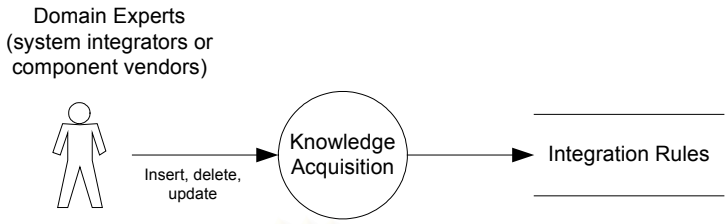

Figure 4-5. Integration Rules Collection

Component attributes define characteristics that impact compatibility with other components, for example, the protocols supported by the component. Integration rules define how attributes affect component integration in terms of compatibility score. These rules identify both those attribute combinations that simplify – and those that complicate – system integration. Table 4-4 identifies integration rules with compatible score level. The higher the score, the better the compatibility.

In practice, each attribute could be assigned different weight to account for individual rule precedence. For example, the system integrator could assign a weight factor of 10 which is the maximum value for function attribute, 9 for platform attribute, and 1 which is the minimum value for language attribute, etc. Hence, the qualified components could be selected by multipling these weight factor (w_i) to the corresponding score level (s_i) of each matched rule. The result would yield a closer compatibility score required by the user. This procedure can be straightforwardly carried out as follows:

$$\text{Compatibility Score} = \sum_{i=1}^{n} (w_i * s_i)$$

where i denotes matched rule i

Due to its inherent arbitrary assignment of weight factor, this research did not incorporate the above procedure into the design of metadata system.

Table 4-4. Integration Rules

Rule Name	Value 1	Value 2	Score level
Language1	JDK 1.1	JDK 1.2	+9
Language2	JDK 1.1	C++	+1
Language3	JDK 1.1	C	+6
Language4	JDK 1.2	JDK 1.2	+10
Language5	JDK 1.1	JDK 1.1	+10
Language6	JDK 1.0	JDK 1.0	+10
Language7	JDK 1.0	JDK 1.2	+7
Language8	JDK 1.0	JDK 1.1	+8
Language9	JDK 1.2	C	+6
Language10	JDK 1.0	C	-10
Language11	JDK 1.2	C++	+1
Language12	JDK 1.0	C++	-10
Platform1	Solaris	Java classes	+10
Platform2	MS Windows	Java classes	+10
Platform3	Solaris libraries	Solaris C Code	+10
Platform4	Windows DLL	Windows binaries	+10
Platform5	Windows DLL	Solaris	-10
Platform6	MS Windows	Solaris libraries	-10
Platform7	UNIX	UNIX	+10
Platform8	UNIX	Mac	+7
RMI_protocol1	RMI 1.1	RMI 1.0	+10
RMI_protocol2	RMI 1.0	RMI 1.0	+10
RMI_protocol3	RMI 1.1	RMI 1.1	+10
RMI_protocol4	RMI 1.2	RMI 1.2	+10
RMI_protocol5	RMI 1.2	RMI 1.1	-10
RMI_protocol6	RMI 1.2	RMI 1.0	-10
GUI1	AWT	AWT	+10
GUI2	Swing	Swing	+10
GUI3	Swing	AWT	-5

These rules specific on component compatibility such as if component A is written in JDK 1.0 and component B is written in JDK 1.1, and weight of language attribute is 7, seven is multiplied by eight points and then added to compatibility score of the ensemble. This score is accumulated until all rules are fired and all components in that ensemble are compared.

2. The Prototype Architectural Design

The two metadata models described above from the basis for an expert-system tool that automates the finding of suitable components and ranking the appropriate set of possible ensembles from these components. In this section, architectural design of the prototype is described with the details of its prototype outlined in the next chapter.

2.1 System Architecture

There are a number of component attributes defined for component integration. In order to verify these attributes of their suitability, a test evaluator system has been designed, as shown in Figure 4-6, and its prototype, described in the next chapter, has been built.

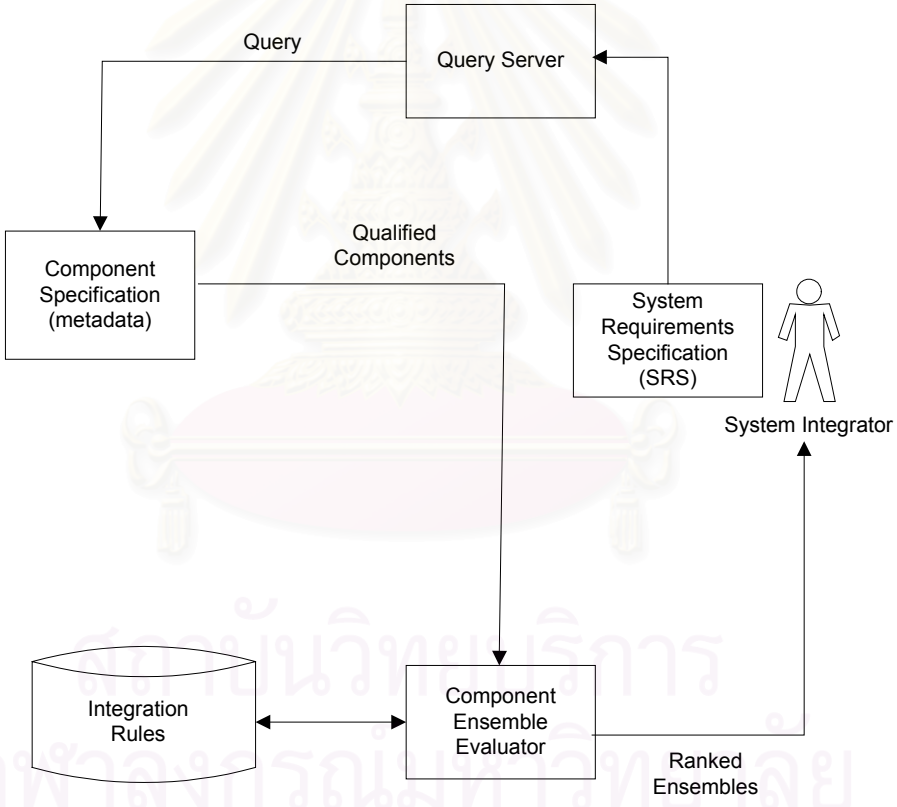

Figure 4-6. System Architecture

The system, as depicted in Figure 4-6, consists of four subsystems in its design. There is a repository of component specifications or metadata that is searchable and already described in section 1.1 above. The integration rule database contains constraints or rules which guide the ensemble evaluator in deciding the appropriateness of each component

ensemble. It is also described in section 1.2 above. There is a query server which accepts requirements specification and queries the repository for suitable components. Lastly, the component ensemble evaluator uses rules in the rule database to evaluate the matching of component selected and ranks the resulting ensembles. All these are driven by the system requirements specification specified by the system integrator. Each subsystem is described in details in the sections that follow.

2.2 System Requirements Specification

The System Requirements Specification or simply SRS consists of one or more component specifications and a set of system constraints. In an actual system this SRS may in fact be simply one element of a larger artifact. System constraints use the same collection of attributes as individual components. For example, Java may be specified as the language of choice for the system. This produces a tension between search constraints and search results and, correspondingly, between requirements and available components. As search constraints are relaxed, additional but less-qualified components will be identified. As additional constraints are added, a smaller group of better-qualified components will be identified. System constraints may compose of language, platform, and function attributes as shown in Figure 4-7.

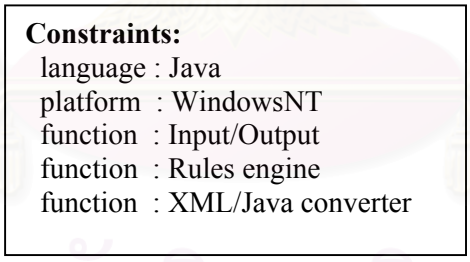

Figure 4-7. A Template for System Requirements Specification

After these constraints are executed, components which written in Java language, run on Windows NT platform, and function Input/Output, Rules engine, and XML/Java converter are identified.

2.3 Component Ensemble Evaluator

Component ensemble evaluation process involves making several trade-off decisions to determine if each candidate component is compatible with other components in the required software system. To make those crucial decisions, information such as

quality attributes of components are necessary. However, most COTS components are delivered as “black box” components; therefore, component interfaces are almost the only source of information available to system developers. Component metadata and integration rules defined in this research provide the information required for evaluation of off-the-shelf components. The component ensemble evaluator uses this information in comparison for component compatibility. It is written in Java programming language. Candidate components have been identified by component selection process as shown in Figure 4-8.

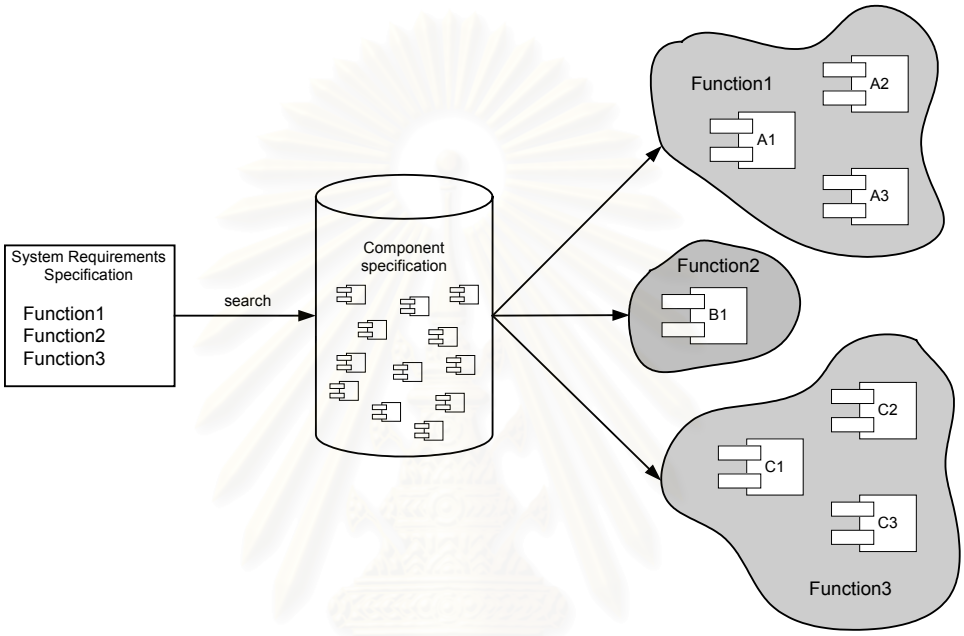

Figure 4-8. Component Selection Process

A sample component selection in Figure 4-8 contains 3 functional requirements: Function1, Function2, and Function3. Three Function1 components: A1, A2, and A3, one Function2 component: B1, and three Function3 components: C1, C2, and C3 are matched the requirements.

The system integrator must discover an *ensemble*—a collection of compatible components— that satisfies the functional requirements. When multiple components are found that match the requirements, however, the number of possible ensembles increases according to the factor of the cardinality of each set of qualified components. This relationship can be formulaically expressed as:

$$\prod_{s \in A} \# \; s$$

where A is the set of component sets.

According to the above requirements and the formula, S consists of

S1 = {A1, A2, A3},

S2 = {B1}, and

S3 = {C1, C2, C3}

Therefore, all possible ensembles (A) are 3 x 1 x 3 = 9 ensembles and can be depicted in Figure 4-9. Those ensembles are:

Ensemble 1 consists of components A1, B1, and C1,

Ensemble 2 consists of components A1, B1, and C2,

Ensemble 3 consists of components A1, B1, and C3,

Ensemble 4 consists of components A2, B1, and C1,

…

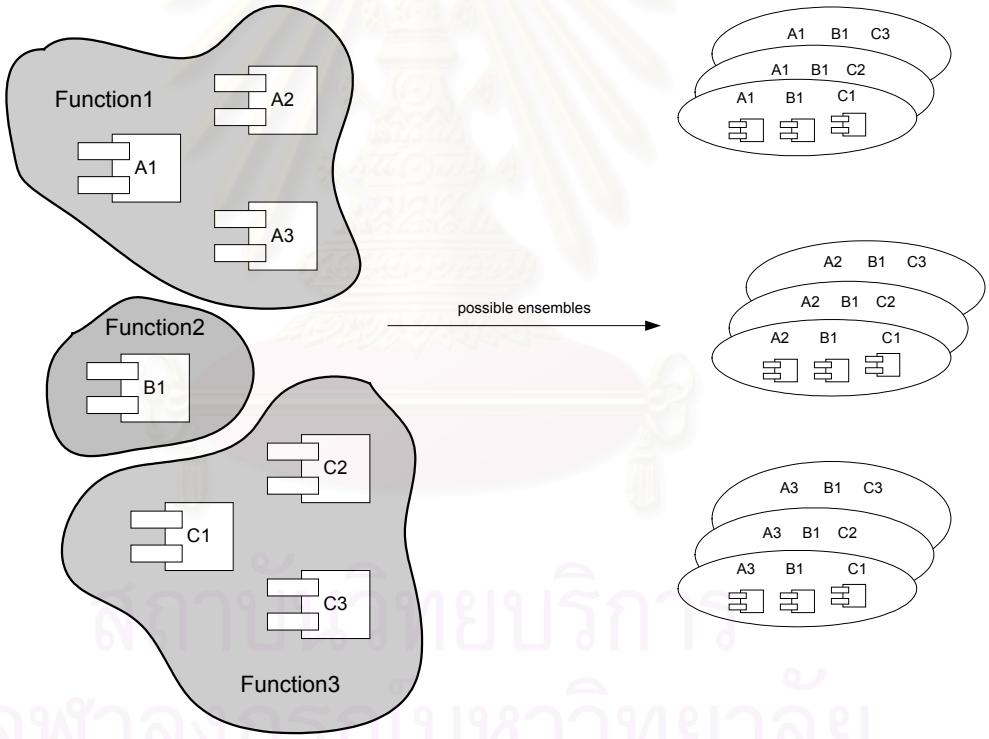

Figure 4-9. Ensemble Formation Process

This file is converted into objects in JRules, which corresponds to actual Java objects. To be evaluated by a rule, the object must exist in working memory. These ensembles are evaluated in working memory one by one resulting in compatibility score. These processes can be shown in Figure 4-10.

It can be seen that there is a danger of exponential explosion of combinations of components into possible ensembles. This exemplifies the importance of Poulin observation (Poulin, 1999). Further investigation is necessary in order to stream line the process and reduce the chance of such a difficulty.

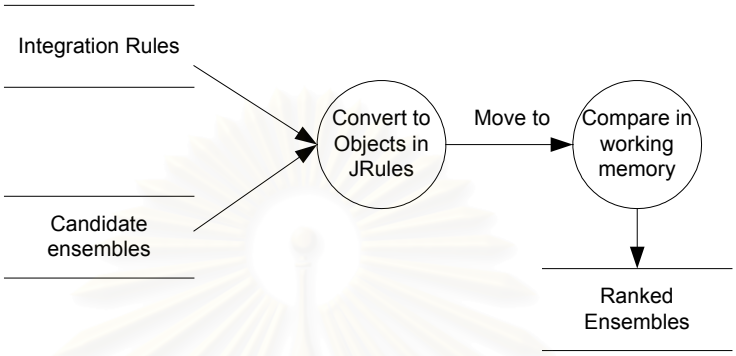

Figure 4-10. Component Ensemble Evaluator Processes

As described above, all databases are designed carefully using many reliable references including domain experts. The implementation details of the prototype constructed from this design are presented in the next chapter.

CHAPTER V

PROTOTYPE

As described in the previous chapter, the prototype has been built to verify the applicability of designed software component attributes. Prototype details are elucidated in the order of implementation below.

1. The Prototype Problem

To build the prototype, a model problem for building a compiler software is assigned. The system integrator specifies the SRS which requires 3 functional components: lexer, parser, and code generator. These requirements are converted into query then compared to component specification. Qualified components are grouped into ensembles and ranked compatibility score using integration rules defined by a domain expert. The result is returned to the system integrator for making further decision. This can be depicted in Figure 5-1.

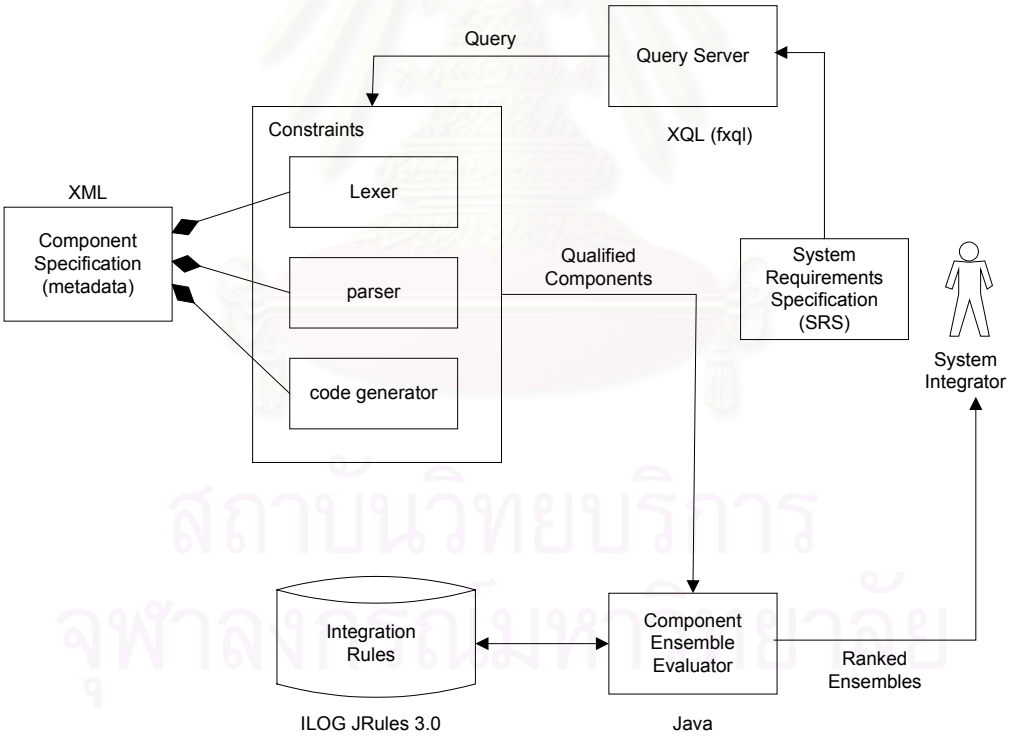

Figure 5-1. System Architecture

Figure 5-1 is the realization of the model presented in the previous chapter. The major components of the prototype are:

1) Component specification,

2) System Requirements Specification (SRS), and

3) Integration rules.

2. Development Environment

This prototype was implemented at the Software Engineering Institute (SEI) of Carnegie Mellon University, Pittsburgh, Pennsylvania, U.S.A. All equipments and licensed softwares were also provided by the SEI. Detail are as follow.

Workstation Pentium III 600 MHz

RAM	256 MB
Harddisk	20 GB
Operating System	WindowsNT 4.0
Rule engine software	ILOG JRules version 3.0
Java compiler	JDK 1.1.8
Script language	Gema
XML Query Language (XQL)	fxql

1.1 Rule engine software

ILOG JRules (ILOG, 2001) provides the expert system engine to drive the integration rules. It is packaged as a set of Java class libraries. As a Java class, the rule engine can be implemented directly, or derived from to add application specific data members and methods. More details are described in integration rules section.

1.2 Java compiler

JDK or Java™ Development Kit (Sun, 2002), contains software and tools that developers need to compile, debug, and run applets and applications written using the Java programming language. The JDK software and documentation is free per the license agreement. JDK 1.1.8 is used because complete support is provided by the Sun Microsystems.

1.3 Script language

Gema (Gray, 1995) is a general purpose text processing utility based on the concept of pattern matching. In general, it reads an input file and copies it to an

output file, while performing certain transformations to the data as specified by a set of patterns defined by the user. It can be used to do the sorts of things that are done by UNIX utilities such as `cpp`, `grep`, `sed`, `awk`, or `strings`. It can be used as a macro processor, but it is much more general than `cpp` because it does not impose any particular syntax for what a macro call looks like. Unlike utilities like `sed` or `awk`, `gema` can deal with patterns that span multiple lines and with nested constructs. It is also distinguished by being able to use multiple sets of rules to be used in different contexts. Gema was selected in this prototype because there were many operations to be completed by the prototype such as formatting text, appending text, Java program running etc. A comparison between Gema, Sed, Awk, and Perl is shown in Table 5-1.

Table 5-1. Comparison of Gema, Sed, Awk, and Perl

	Gema	**Sed**	**Awk**	**Perl**
Regular expressions	x	x	x	x
Non-line-oriented	x			
Non-procedural and rule-based	x		x	
Recognizers	x			x
User-defined recognizer	x			
Unification of matching and function definition	x			
Multiple rule sets	x			
Context-sensitive matching	x			
Recursive patterns	x			
Dynamic patterns	x			
User-specified match position	x			
User-definable syntax	x			
General-purpose programming	x		x	x
OS Interface	x			x
Command-line extension	x			x

1.4 XML Query Language (XQL)

The XQL (Wattle, 2000) is a notation designed to address XML documents just as SQL is used to access relational databases. There are a number of functions to

the XQL. XML Document Matching examines if XML documents match XQL query statements. XML Document Query retrieves data from XML documents according to XQL query statements. Query Result Output dispatches XQL query results to specified destinations. To identify all the components in the component repository that satisfy the SRS, the set of constraints is extracted from the SRS and transformed into XQL (Robie, 1999) queries. Although XQL is one of several XML query language proposals that enjoys the support of several commercial products.

Once constructed, XQL queries are run against the component repository to identify a set of candidate components. For example, the SRS may specify a functional requirement for a spell checker component as shown below:

<function>spell checker</function>

This would be extracted as an XQL query as follows:

"//component/general_info/function[spell checker]"

Running this query against the component repository generates a working set of components that implement this functionality. A subset of XQL called fxql is used for the same command format in this prototype.

3. Component Specification

The component specification is represented in XML. It is well suited for this application as it provides a formal language for mapping values to attributes and is fully extensible.

```
<?xml encoding="UTF-8"?>
<!DOCTYPE components SYSTEM “component.dtd”>
<!-- Root element contains one or more components-->
<!ELEMENT components (component+)>
<!ELEMENT component (general_info, protocol, security)>
<!ATTLIST component cpid ID #REQUIRED>
<!ELEMENT general_info (name,  version,  vendor, platform+, function+, framework, language,
space_req,  domain+,  keywords, gui? , cost, license, lang_support+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
…
```

Figure 5-2. A Part of Component Specification DTD

Root element in Figure 5-2 is *components* which may consist of one or more element *component*. The *component* element contains component identifier (cpid) and 3 sub-elements: general_info, protocol, and security elements. Element general_info contains 14 sub-elements: name, version, vendor, platform, function, framework, language, space_req, domain, keywords, gui, cost, license, lang_support. Element: name, version, and platform are character data type, and so on.

Full component specification in DTD and XML documents are shown in Appendix A.

4. System Requirements Specification (SRS)

The system requirements specification is also represented in XML. Figure 5-3 shows SRS DTD which root element is constraint, system integrator can specify only functional requirement since other attributes are optional.

```
<?xml encoding=”UTF-8”?
<!DOCTYPE constraint SYSTEM “SRS.dtd”>
<!ELEMENT constraint (function+, platform?, language?, protocol?)>
<!ELEMENT function (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
<!ELEMENT language (#PCDATA)>
<!ELEMENT protocol (#PCDATA)>
```

Figure 5-3. System Requirements Specification DTD

Attributes specified in the SRS for individual components similarly limit component candidates to those that match the requirement. Figure 5-4 is an XML document of the SRS which contains 3 functional requirements of compiler software system.

```
?xml version=”1.0” encoding=”UTF-8”?>
<?xml-stylesheet type=”component/xsl” href=”component.xsl”?>
<constraint>
  <function>Lexer</function>
  <function>parser</function>
  <function>code_generator</function>
</constraint>
```

Figure 5-4. SRS in XML document

The ability to modify requirements and re-execute a search is a major benefit of automating this process. This allows system integrators to adjust system requirements to accommodate changes in market realities in real time.

5. Integration Rules

After required components are selected from component specification repository, their characteristics are compared for compatibility using integration rules. Figure 5-5 depicts the process of knowledge acquisition from a domain expert.

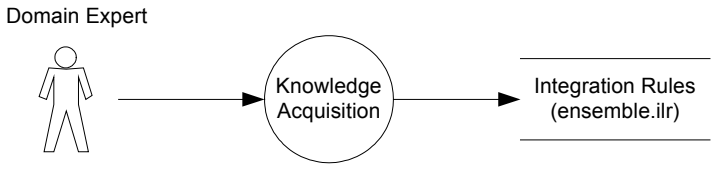

Figure 5-5. Data Flow Diagram of Creating Integration Rules

The rules used in this prototype are extracted from the senior technical researcher at the SEI, Mr. Robert C. Seacord, who has 17 years of software development experience in industry, defense, and research. His principal areas of expertise includes component-based development, graphical interface design, human factors. He defines integration rules for component integration and score level of compatibility. Compatibility score of +10 means those two components have high possibility in working well together.

Before starting this prototype, the trial version of rule engine software named "JRules" from ILOG company was tried for thirty days and found that it is suitable for implementation because it is easy to use, and is an object-oriented rule-based programming language in which XML document can be converted into Java objects and compared with integration rules in the same format.

ILOG JRules is a rich and flexible product aimed at enabling software developers to create applications that can be maintained with minimal effort. It allows developers to combine rule-based and object-oriented programming to add business rules to new and existing applications. It fully supports JDKv1.1 through J2SDKv1.3, J2SE, and J2EE. A JRules application consists of a set of rules and a collection of objects. Each rule is composed of three parts: a header, a condition part, and an action part as shown in Figure 5-6.

```
rule ruleName { (priority = value; )
  ( packet = packetName; )

when {  conditions …}

then  {  [ actions …] }

};
```

Figure 5-6. ILOG JRules Rule Structure

The header defines the name of the rule, its priority and packet name. It starts with the keyword **rule**. The condition part begins with the keyword **when**, and is also referred to as the left-hand side (LHS) of the rule. It defines the conditions that must be met in order for the rule to be eligible for execution. The action part, which begins with the keyword **then**, is referred to as the right-hand side (RHS) of the rule. When all of the LHS conditions are met, the RHS of the rule is executed (or 'fires') (ILOG, 2001). Objects in JRules correspond to actual Java objects. To be evaluated by a rule, the object must exist in working memory. Placing an object in working memory is accomplished in JRules through the use of an **ASSERT** statement.

A sample rule for evaluating language compatibility is shown in Figure 5-7. This rule evaluates language compatibility between components written in Java, JDK 1.1 and JDK1.2. In this example, if component **`?c1`** is implemented using JDK1.1 and component **`?c2`** is implemented in JDK1.2, nine points are added to the compatibility score for the ensemble. This value is added because a well-defined interface exists between JDK1.1 and JDK1.2.

```
rule LanguageCompatible1
      {
          priority = high;
          when
            {
              ?c1: Component( lang.equals("JDK1.1"); ?i1:id);
              ?c2: Component( lang.equals("JDK1.2"); ?i2:id ; ?i1 !=
      ?i2);

              ?e: Evaluation();

            }
          then
            {

              modify ?e {score += 9;}

            }
      };
```

Figure 5-7. Language Compatibility Rule

All integration rules are shown in Appendix B.

6. Structure of the Prototype

All processes and files in implementation can be described using data flow diagram as shown in Figure 5-8. Line with arrow specifies sequence of processes and/or direction of input/output file. All processes are described as follows:

1) System integrator specifies a *system requirements specification* (SRS.xml) that incorporates elements of the system requirements and architecture, as well as creates *component specification* (component.xml) that describes each component and also creates integration rule file (ensemble.ilr).
2) *Required components* are matched with *component specification* in a repository with component selection process.
3) Components are grouped into *ensembles* (ensemble.xml).
4) Ensembles are evaluating using *integration rules* (ensemble.ilr) in the knowledge-base.
5) *Ranked ensembles* are returned to the system integrator.

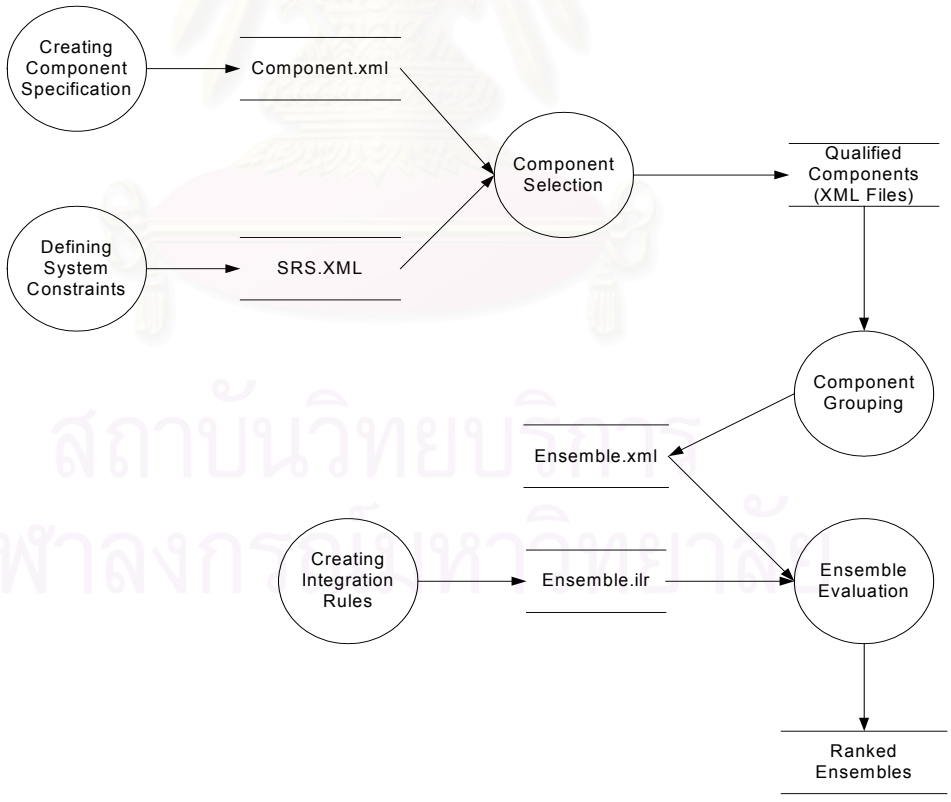

Figure 5-8. Data Flow Diagram

7. Populating the Prototype

Required components in the component repository are populated in the manner as described in section 7.1. Qualified components are populated to form an ensemble as described in section 7.2.

7.1. Component Selection

Constraints are extracted from the system requirements specification and converted to XQL queries. These queries are run against the component specification repository to identify a set of candidate components.

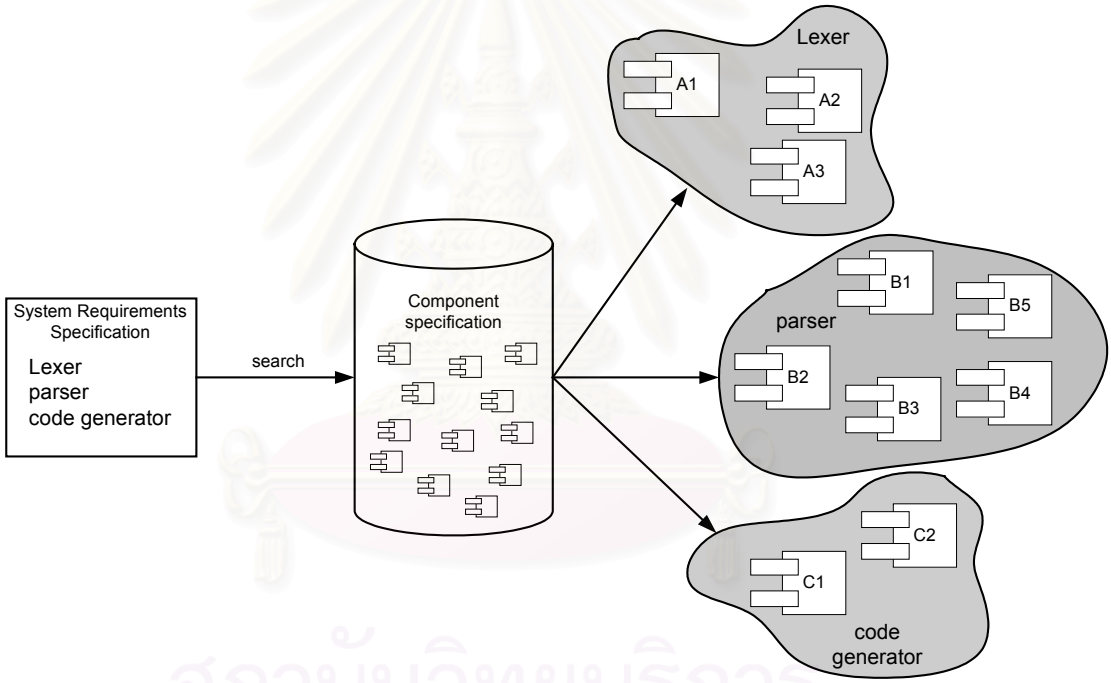

Figure 5-9. Component Selection

The SRS consists of 3 functional requirements: Lexer, parser, and code generator. The component specification repository contains 3 Lexers: A1, A2, and A3, 5 parsers: B1, B2, B3, B4, and B5, and 2 code generators: C1, and C2 as shown in Figure 5-9.

7.2. Ensemble Formation

As described in the previous chapter, ensembles are formed according to the formula as shown in Figure 5-10.

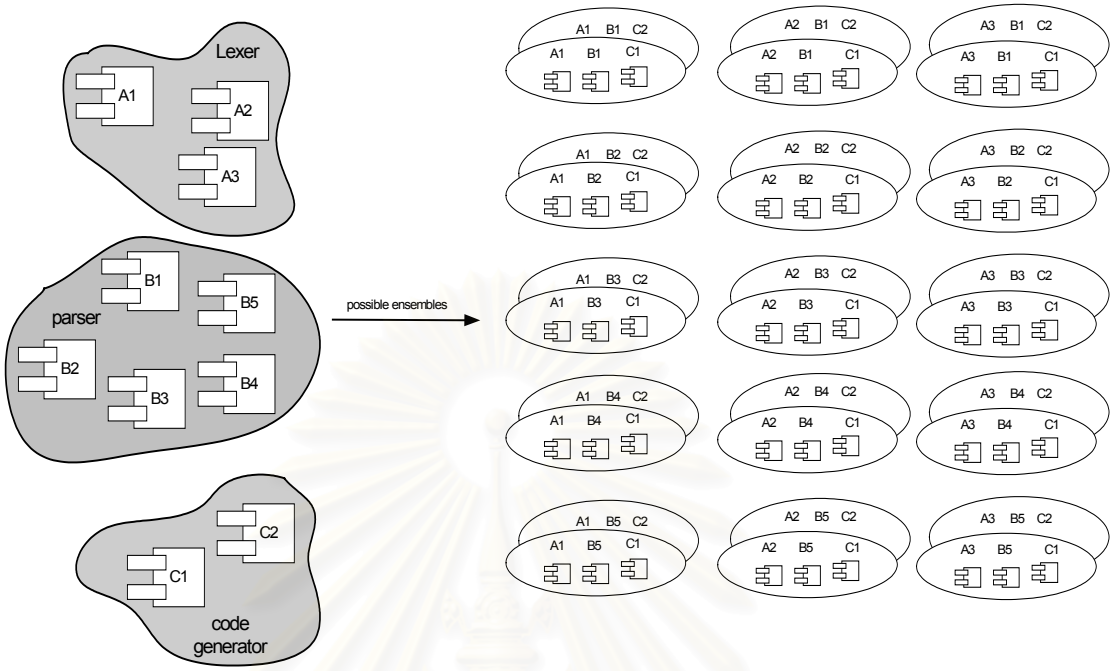

Figure 5-10. Ensemble Formation

8. Result

In the prototype implementation, the component specification contains 3 lexers, 5 parsers, and 2 code generators. Therefore, all candidate ensembles that meet the requirements are 3 x 5 x 2 = 30 ensembles. Each ensemble is compared to integration rules as described in the previous section. The result is compatibility score and it is accumulated until all components in that ensemble are performed. Table 5-2 shows ensemble compatibility scores of 26 ensembles in descending sequence including component identifiers of each ensemble. These ensembles are selected from a component repository containing 3 lexers (component IDs: 100, 115, 210), 5 parsers (component IDs: 105, 106, 107, 109, 1001) and 2 code generators (component IDs: 101, 103).

Table 5-2. Ensembles Compatibility Scores

Ensemble	*Component IDs*	*Score*
21	(115 101 107)	39
20	(210 101 107)	39
8	(210 101 106)	29
26	(210 101 105)	29
24	(115 101 105)	29
3	(115 103 106)	20
10	(115 101 1001)	20
11	(100 103 106)	18
4	(100 101 105)	14
6	(100 101 106)	11
2	(100 103 107)	11
7	(210 103 100)	10
5	(100 103 1001)	1
9	(100 101 1001)	-5
1	(100 101 109)	-15

Ensembles 20 and 21 both share a high score of 39. These scores resulting from the application of the integration rules to the set of attributes defined for the components included in each ensemble. Components in both of the highly ranked ensembles share a number of attributes, and the remaining attributes are not highly incompatible, resulting in relatively high overall scores for the ensembles.

In this chapter, all implementation details and result of the prototype are discussed and described. Conclusion and future work are discussed in the next chapter.

CHAPTER VI

CONCLUSION AND FUTURE WORK

It is apparent from prototype verification that the proposed system will serve as software component repository. The contribution will be utmost important to software reuse and component-based development. The conclusion of this research and future work are discussed in the sections that follow.

1. Conclusion

A metadata model of software components has been defined and constructed to be used by component developers and users to describe and select the required software components. Component selection is one of the key activities in component-based software development to ensure that appropriate components are selected for integration when constructing high-quality software systems. If the component structure or metadata in the component repository is a well-defined structure, components can be found and reused effectively.

The prototype so developed for this research presents an ensemble compatibility evaluation using defined software component metadata and a set of integration rules. The overall system was designed and a prototype built to evaluate the concept. The prototype includes the construction of component specification or metadata, system requirements specification, and integration rule databases.

Component specification or metadata describes attributes of each component and is in the form of an XML document that can be created by any text editor or specific XML editor. It contains all software component information necessary for component integration such as communication protocol, written language, platform, framework, etc.

System requirements specification, provided by system integrators, defines system constraints. Attributes specified in the system requirements specification for individual components limit components candidates to those that match the requirement. The system requirements specification is also an XML document and is converted to an XQL command prior to the component selection process.

Integration rules define how attributes affect component integration. To implement these rules, the rule engine software called ILOG JRules software, was utilized and used. The integration rules are kept in a ".ilr" file and called by the rule engine software to compare how compatible the components are. The compatibility score is accumulated until all rules are fired. The results are returned to system integrators for further consideration.

The prototype allows system integrators to explore a broader component space than what is possible when using manual techniques. It also quickly eliminates components that are overly difficult to integrate.

In addition to aiding in the evaluation of component ensembles, the prototype provides a mechanism for preserving, sharing and re-using hard-won system integration knowledge. System integrators can use this information to identify compatible ensembles of components and actively expand upon it as additional insights into the rules that govern system integration have emerged.

The greatest challenge of this model is not the feasibility of automating the process, but the ability to collect the data necessary to drive the process. First, it is required to populate the component repositories with a sufficient number of component specifications to guarantee that an SRS can be satisfied from the pool of available components. Second, it is required to identify, through successive rounds of refinement, the attributes that are used to describe each component and the set of system integration rules that are used to compute the compatibility of ensembles.

It is hoped that component brokers, who market third-party components, may initially provide component search engines based on the model. Component consumers will benefit from this service to discover components that can be easily integrated into their systems and begin to rely on these component brokers. Component producers may then feel compelled to define component specifications that are compatible with this approach. Hopefully, this process will culminate in a standard component repository and component specification format.

Although this research contributes a software component metadata model to software engineering community especially component-based software development approach, there are some limitations in the prototype that should be modified or added as follows:

- Integration rules come from only one domain expert in component-based software development at the SEI. If rules come from several domain experts, various kinds of compatibility scores can be expanded.
- Only functional requirements of the system constraints are considered in this prototype. This may cause a vast number of candidate components in the ensembles. It is expected that if various aspects of the constraints are specified, the smaller the group of better-qualified candidates will be identified.

2. Future Work

There are many significant work left to be achieved before a robust ensemble evaluator can be produced. Many future research areas are possible:

- Implementation with real world component models such as EJB, CORBA, or DCOM to evaluate this approach against their characteristic.
- Scale up this prototype by expanding component specification repository size. But vast numbers of candidate components will be involved in the evaluation and can result in exponential explosion of possibilities.
- Integration rule extension. As discussed in previous section: the more rules we have, the more categories of compatibility are measured. The main problem is the lack of experienced system integrators or domain experts in Thailand where this research is performed. We can solicit expert opinion by broadcasting the questionnaires on the Internet.
- Component ensemble feedback database. From ranked compatibility score of each ensemble set which has been suggested to the system integrator, it is believed that if an ensemble set is implemented by different system integrators, the results of the software systems are not the same. As the result depends on the capability and experience of the system integrator in defining an environment. Therefore, it is very useful for future system integrators if experienced integrators record all relevant information on working environments of integration. In this way, system integrators who

want to use previous ensemble set can review the comments on integration of the ensemble set from the feedback database.

For the last two approaches, experiences from system integrators play an important role in collecting data. To try this, a number of software development companies in Bangkok have been contacted for an interview about component-based software development in Thailand. Unfortunately, the information from the interviews is not enough for further research because software companies in Thailand tend to use in-house components rather than components from other vendors, to develop application software systems. Their reasons are as follows:

- Buying components is prohibitively expensive. This not only involves a one-time cost, but continuous expenses such as license term and maintenance contract must be considered,
- Labor wages in Thailand is quite low compared to post-industrialized countries like the United States. It is not too difficult to find good programmers, and
- Most components in the market do not meet developers' requirements so it is better to build their own.

Table 6-1 lists the three companies that provide information on the states of the component market.

Table 6-1. List of Interviewed Companies

Company Name	Business Activity
IBM Corperation (Thailand)	System integrator / Software developer / Consultant / Programming/ Database / Embedded Systems
ProSolutions Asia Pacific Co., Ltd.	System integrator / Software developer / Consultant / Programming/ Database / Embedded Systems
PSP (Thailand) Co., Ltd.	System integrator / Software developer / Consultant / Programming/ Database / Embedded Systems / Software Value Added Reseller

Preliminary interviews have been conducted where a number of companies were contacted. It was found that there were many various reasons to refuse the interview. Thus, the information given by companies listed in Table 6-1 and 6-2 may not reflect the true needs of commercial sector.

Table 6-2. List of Other Contacted Companies.

Company Name	Business Activity
BizCuit Co., Ltd.	System integrator / Software developer / Consultant / Programming
Computer Science Corporation Limited	Software Developer / Programming / Database / Web Authoring and Design
Data Express Co.,Ltd.	System Integrator / Consultant / Software Value Added Reseller, Dealer, or Distributor
Express Software Group Co., Ltd.	Software Developer/ Programming / Database
Golden International Information Co., Ltd.	System integrator / Software developer / Consultant / Programming/ Database
Headway Technology Co., Ltd.	Software Developer/ Programming / Database
System Plus Group Co., Ltd.	System integrator / Software developer / Consultant / Programming/ Database / Web Authoring and Design
Ultimax Co., Ltd.	System integrator / Software developer / Consultant / Programming/ Database

It is quite apparent from this effort that CBSE, especially in low labor-wage countries, is still in its infancy and it will be difficult for some time to find sufficient empirical data concerning the use of CBSE enough to populate the model in order to evaluate its accuracy and usefulness. Nevertheless, the model built here will contribute to the future efforts in organizing and using components in CBSE.

REFERENCES

Anderson, R. et. al. Professional XML. (n.p.): Wrox Press, 2000.

Bachmann, Felix; Bass, Len; Buhman, Charles; Comella-Dorda, Santiago; Long, Fred; Robert, John; Seacord, Robert; & Wallnau, Kurt. Technical Concepts of Component-Based Software Engineering, Volume II. (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, 2000.

Basili, V. R. and Boehm, B. COTS-Based Systems Top 10 List. IEEE Computer 34, 5 (May 2001) : 91-93.

D'Souza, D. and Wills, A.C. Objects, Components and Frameworks with UML: The Catalysis Approach. Reading: MA: Addison-Wesley, 1999.

Chang, D. and Harkey, D. Client/Server Data Access with Java and XML. (n.p.): Wiley, 1998.

Clark, James. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, Available from: http://www.w3.org/TR/xslt. visited date: 11/ 1999.

Component Registry Company. Available from: http://www.ComponentRegistry.com. visited date: 10/2001.

Dornfest, Rael and Brickley, Dan. The Power of Metadata. Available from: http://www.openp2p.com/pub/a/p2p/2001/01/18/metadata.html. visited date: 01/2002.

Garlan, D.; Allen, R.; and Ockerbloom, J. Architectural Mismatch: Why Reuse is So Hard. IEEE Software. 12, 6 (Nov.1995): 17-26.

Gorman, Michael. Metadata or Cataloguing? A False Choice. Journal of Internet Cataloging. 2, 1 (1999): 5-22.

Gray, D. N. gema - the general purpose macro processor. Available from: http://www.ugcs.caltech.edu/gema. April, 1995.

Gulbransen, D. The Complete IDIOT'S Guide to XML. QUE, 2000.

Henning, M. and Vinoski, S. Advance CORBA Programming with C++. Reading: Addison-Wesley, MA. 1999.

Heineman, George T. and Councill, William T. Component-Based Software Engineering Putting the Pieces Together. (n.p.): Addison-Wesley, 2001.

Hurwitz, Judith. Preparing for Component-Based Development. white paper, March 1998.

ILOG Company. ILOG JRules White paper ILOG. Available from: http://www.ilog.com. May 2001. visited date: 02/2002.

Kobryn, Cris. Modeling Components and Frameworks with UML. Communications of the ACM. Vol. 43 No. 10 (October 2000): 31-38.

Luger, George F. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Fourth Edition. (n.p.): Addison-Wesley, 2002.

Mili, Rym; Mili, Ali; & Mittermeir, Roland T. Storing and Retrieving Software Components: A Refinement Based System. IEEE Transaction on Software Engineering . 23, 7 (July 1997) : 445-460.

Maurer, Peter M. Components: What if they gave a revolution and nobody came?. IEEE Computer . 33, 6 (June 2000) : 28-34.

Microsoft: Available from: http://www.microsoft.com/isapi. Mar 2000.

Microsoft: Available from: http://www.microsoft.com/net/. 2002.

Morisio, M. et al. Investigating and Improving a COTS-Based Software Development Process. Proceedings of the 22nd international conference on Software Engineering June 2000 : 31-40.

Mowbray, T. J. and Ruh, W. A. Inside CORBA Distributed Object Standards and Applications. (n.p.): Addison-Wesley, 1997.

Murphy, Lisa D. Digital Document Metadata in Organizations: Roles, Analytical Approaches, and Future Research Directions. Proceedings of the Thirty-First Hawaii International Conference on System Science (1998): 267-276.

Object Management Group. The Common Object Request Broker: Architecture and Specification. Version 2.4, 2000.

OMG: Available from: http://www.omg.org/corba/whatiscorba.html. Mar 2000.

Ogbuji, Uche. RIL: A Taste of Knowledge. Available from: http://www.xml.com/pub/2000/10/11/rdf/ril.html. October 2000.

Penix, John et al. Automating Component Integration for Web-Based Data Analysis. Proceedings of IEEE Aerospace Conference. (2000).

Poulin, Jeffrey S. Reuse: Been There, Done That. Communications of the ACM Vol. 42 No. 5 (May 1999) : 98-100.

Poulin, Jeffrey S. and Yglesias, Kathryn P. Experiences with a Faceted Classification Scheme in a Large Reusable Software Library (RSL). Proceedings of Computer Software and Applications Conference COMPSAC 93, 1993.

Pour, Gilda. Component-Based Software Development Approach: New Opportunities and Challenges. Proceedings of Technology of Object-Oriented Language (TOOLS 26) 1998 : 376-383.

Pour, Gilda. Enterprise JavaBeans, JavaBeans & XML expanding the Possibilities for Web-Based Enterprise Application Development. Proceedings of Technology of Object-Oriented and Systems (1999) : 282-291.

Pour, Gilda; Griss, Martin L; Lutz, Michael. The Push to Make Software Engineering Respectable. IEEE Computer. 33, 5 (May 2000) : 35-43.

Poulin, Jeffrey S. and Werkman, Keith J. Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components. Proceedings of the 17th international conference on software engineering on Symposium on software reusability. (1995) : 160-168.

Prieto-Diaz, R. & Freeman, P. Classifying Software for Reusability. IEEE Software. 4, 1 (January 1987) : 6-16.

Robie, Jonathan. XQL (XML Query Language). Available from: http://www.ibiblio.org/xql/xql-proposal.html. August, 1999.

Roy, J. and Ramanujan, A. Understanding Web Services. IT Professional. 3, 6 (Nov. – Dec. 2001) : 69-73.

Sauer, Ly Danielle et al. Meta-Component Architecture for Software Interoperability. Proceedings of International Conference on Software Methods and Tools SMT 2000 : 75-84.

Seacord, Robert C.; Wallnau, Kurt; John, Robert; Comella-Dorda, Santiago; & Hissam, Scott A. Custom vs. Off-the-Shelf Architecture. Proceedings of the 3rd International Enterprise Distributed Object Computing Conference. Mannheim, Germany, September 27-30, 1999.

Seacord, Robert C.; Hissam, Scott A.; Wallnau, Kurt C. AGORA: A Search Engine for Software Components. IEEE Internet Computing. (November-December 1998) : 62-70.

Sun Microsystems. Available from: http://java.sun.com/products/javabeans/docs/. and http://java.sun.com/products/ejb/. Mar, 2000.

Sun Microsystems. Available from: http://java.sun.com/products/jdk/1.1. 2002.

Szyperski, Clements. Component Software: Beyond Object-Oriented Programming. (n.p.): Addison-Wesley, 1997.

Traas, Vincent and Hillegersberg, Jos Van. The Software Component Market on the Internet Current Status and Conditions for Growth. ACM SIGSOFT, Software Engineering Notes. 25, 1, January 2000 : 114-117.

Wallnau, K. C.; Hissam, S. A.; Seacord, R. C. Building Systems from Commercial Components. (n.p.): Addison-Wesley, 2002.

Wang, Nanbor; Schmidt, D. C.; and O'Ryan, Carlos. Overview of the CORBA Component Model. Component-Based Software Engineering: Putting the pieces together. (n.p.): Addison-Wesley, 2001.

Wattle Software. Available from: http://www.xmlwriter.net/user_tools/fxql.shtml. Sept., 2000.

Yakimovich, Daniil; Bieman, James M.; Basili, Victor R. Software architecture classification for estimating cost of COTS integration. Proceedings of the 1999 international conference on Software Engineering. May 1999.

Yau, Stephen S. and Dong, Ning. Integration in Component-Based Software Development Using Design Patterns. Proceedings of Computer Software and Applications Conference. 2000 : 369-374.

APPENDICES

APPENDIX A

The component specification is divided into 2 parts: Document Type Definition and XML document which describes component metadata. The first is a file “component.dtd” as shown below:

```
<?xml encoding="UTF-8"?>
<!-- Root element contains one or more components-->
<!ELEMENT components (component+)>
<!-- each component has 3 group elements-->
<!--general_info, protocol, and security-->
<!ELEMENT component (general_info,
                     protocol,
                     security)>
 <!ATTLIST component cpid ID #REQUIRED>
<!ELEMENT general_info (name,
                        version,
                        vendor,
                        platform+,
                        function+,
                        framework,
                        language,
                        space_req,
                        domain,
                        keywords,
                        gui,
                        cost,
                        license,
                        lang_support+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
<!ELEMENT function (#PCDATA)>
<!ELEMENT framework (#PCDATA)>
```

```
<!ELEMENT language (#PCDATA)>
<!ELEMENT domain (#PCDATA)>
<!ELEMENT space_req (disk_space,
                memory_space)>
<!ELEMENT disk_space (#PCDATA)>
  <!ATTLIST disk_space unit ID #REQUIRED>
<!ELEMENT memory_space (#PCDATA)>
  <!ATTLIST memory_space unit ID #REQUIRED>
<!ELEMENT keywords (keyword+)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT gui (#PCDATA)>
<!ELEMENT cost (#PCDATA)>
<!ELEMENT license (#PCDATA)>
<!ELEMENT lang_support (#PCDATA)>

<!ELEMENT vendor (name,
                        phone,
                        address,
                        url,
                        contact)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT address (street,
                         city,
                         state,
                          zip)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT contact (email+)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT protocol (name, version, provider, RMI_protocol)>
```

```
  <!ATTLIST protocol credential CDATA #REQUIRED>
<!ELEMENT RMI_protocol (#PCDATA)>
<!ELEMENT provider (name,
                    phone,
                    address,
                    url,
                    contact)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT security (confidentiality?,
                   authentication?,
                   nonrepudiation?)>
<!ELEMENT confidentiality (name,  provider)>
  <!ATTLIST confidentiality credential CDATA #REQUIRED>
<!ELEMENT authentication (name, provider)>
  <!ATTLIST authentication credential CDATA #REQUIRED>
<!ELEMENT nonrepudiation (name, provider)>
  <!ATTLIST nonrepudiation credential CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT provider (name,
                    phone,
                    address,
                    url,
                    contact)>
```

The second part is component XML document. It contains component metadata, its file name is component.xml. A component metadata file is so large, so only three metadata of components are represented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="component/xsl" href="component.xsl"?>
<!--DOCTYPE components SYSTEM "component.dtd"-->
<!--This is XML document for software component uses component.dtd-->
<components xmlns="x-schema:ComponentsMSSchema.xml">
  <component cpid="100">
```

```
<general_info>
 <comp_name>JLex</comp_name>
 <version>1.2.5</version>
 <vendor>
  <name>abc</name>
  <phone>412-455-6233</phone>
  <address>
   <street>250 Fifth Ave.</street>
   <city>Pittsburgh</city>
   <state>Pennsylvania</state>
   <zip>15213</zip>
  </address>
  <url>www.abc.com</url>
  <contact>
   <email>jim@abc.com</email>
  </contact>
 </vendor>
 <platform>Unix</platform>
 <function>Lexer</function>
 <framework>CORBA</framework>
 <language>JDK1.2</language>
 <space_req>
  <disk_space unit="bytes">1200</disk_space>
  <memory_space unit="MB">50</memory_space>
 </space_req>
 <domain>general</domain>
 <keywords>
  <keyword>sequence</keyword>
  <keyword>sort</keyword>
 </keywords>
 <gui>Swing</gui>
 <cost unit=”$”>995</cost>
 <license>on_agreement</license>
 <lang_support>English</lang_support>
```

```
  <lang_support>Thai</lang_support>
</general_info>
<protocol credential="provider">
  <name>IIOP</name>
  <version>2.0</version>
  <provider>
    <name>Inprise</name>
    <phone>1-800-123-5213</phone>
    <address>
      <street>101 Number One Street</street>
      <city>Norwell</city>
      <state>Massachusetts</state>
      <zip>02061</zip>
    </address>
    <url>www.inprise.com</url>
    <contact>
      <email>info@inprise.com</email>
    </contact>
  </provider>
  <RMI_protocol>RMI1.2</RMI_protocol>
</protocol>
<security>
  <confidentiality credential="testing">
    <name>SSL</name>
    <provider>
      <name>Inprise</name>
      <phone>1-800-123-5213</phone>
      <address>
        <street>101 Number One Street</street>
        <city>Norwell</city>
        <state>Massachusetts</state>
        <zip>02061</zip>
      </address>
      <url>www.inprise.com</url>
```

```
      <contact>
        <email>info@inprise.com</email>
      </contact>
    </provider>
  </confidentiality>
 </security>
</component>

<component cpid="101">
 <general_info>
  <name>comm_pro</name>
  <version>1.0</version>
  <vendor>
   <name>communication technology company</name>
   <phone>301-455-4623</phone>
   <address>
    <street>250 Number Two Ave.</street>
    <city>Mytown</city>
    <state>Pennsylvania</state>
    <zip>15213</zip>
   </address>
   <url>www.communicate.com</url>
   <contact>
    <email>support@communicate.com</email>
   </contact>
  </vendor>
  <platform>Windows DLL</platform>
  <function>code_generator</function>
  <framework>CORBA</framework>
  <language>JDK1.1</language>
  <space_req>
      <disk_space unit="bytes">1650</disk_space>
      <memory_space unit="MB">90</memory_space>
  </space_req>
```

```
  <domain>communication</domain>
  <keywords>
    <keyword>Internet</keyword>
    <keyword>network</keyword>
  </keywords>
   <gui>AWT</gui>
  <cost unit=”$”>1350</cost>
  <license>one-time</license>
  <lang_support>English</lang_support>
  <lang_support>Japanese</lang_support>
</general_info>
<protocol credential="provider">
  <name>TCP/IP</name>
  <version>3.0</version>
  <provider>
    <name>Net service</name>
    <phone>1-800-643-7213</phone>
    <address>
      <street>1245 Number Two Street</street>
      <city>Norwell</city>
      <state>Massachusetts</state>
      <zip>02061</zip>
    </address>
    <url>www.inprise.com</url>
    <contact>
      <email>info@inprise.com</email>
    </contact>
  </provider>
  <RMI_protocol>RMI1.0</RMI_protocol>
</protocol>
<security>
  <confidentiality credential="testing">
    <name>Kerboros</name>
    <provider>
```

```
      <name>Inprise</name>
      <phone>1-800-123-5213</phone>
      <address>
        <street>101 Number Two Street</street>
        <city>Norwell</city>
        <state>Massachusetts</state>
        <zip>02061</zip>
      </address>
      <url>www.inprise.com</url>
      <contact>
        <email>info@inprise.com</email>
      </contact>
    </provider>
  </confidentiality>
 </security>
</component>

<component cpid="105">
 <general_info>
  <name>XParser</name>
  <version>2.2</version>
  <vendor>
    <name>Parser technology company</name>
    <phone>301-455-4623</phone>
    <address>
      <street>250 Number Two Ave.</street>
      <city>Mytown</city>
      <state>Pennsylvania</state>
      <zip>15213</zip>
    </address>
    <url>www.communicate.com</url>
    <contact>
       <email>support@communicate.com</email>
    </contact>
```

```
  </vendor>
  <platform>Windows</platform>
  <function>parser</function>
  <framework>ActiveX</framework>
  <language>JDK1.2</language>
  <space_req>
   <disk_space unit="bytes">1247</disk_space>
   <memory_space unit="MB">850</memory_space>
  </space_req>
  <domain>Web application</domain>
  <keywords>
   <keyword>XML</keyword>
   <keyword>Parser</keyword>
  </keywords>
  <gui>Swing</gui>
  <cost unit=”$”>860</cost>
  <license>one-time</license>
  <lang_support>English</lang_support>
</general_info>
<protocol credential="provider">
  <name>TCP/IP</name>
  <version>3.0</version>
  <provider>
   <name>Net service</name>
   <phone>1-800-643-7213</phone>
   <address>
    <street>1245 Number Two Street</street>
    <city>Norwell</city>
    <state>Massachusetts</state>
    <zip>02061</zip>
   </address>
   <url>www.inprise.com</url>
   <contact>
     <email>info@inprise.com</email>
```

```
      </contact>
    </provider>
    <RMI_protocol>RMI1.1</RMI_protocol>
  </protocol>
  <security>
    <confidentiality credential="testing">
      <name>Kerboros</name>
      <provider>
        <name>Inprise</name>
        <phone>1-800-123-5213</phone>
        <address>
          <street>101 Number Two Street</street>
          <city>Norwell</city>
          <state>Massachusetts</state>
          <zip>02061</zip>
        </address>
        <url>www.inprise.com</url>
        <contact>
            <email>info@inprise.com</email>
       </contact>
      </provider>
    </confidentiality>
  </security>
 </component>
</components>
```

APPENDIX B

All 27 representative integration rules extracted from the domain expert as described in Chapter 5 in the form of ILOG JRules rule structure are shown below.

```
import component.*;

setup
{
   assert Evaluation() {
         score = 0; }


};


rule LanguageCompatible1
{
   priority = high;
   when
    {
         ?c1: Component( lang.equals("JDK1.1"); ?i1:id);
         ?c2: Component( lang.equals("JDK1.2"); ?i2:id ; ?i1 != ?i2);

         ?e: Evaluation();

    }
   then
    {

         modify ?e {score += 9;}

         System.out.print("\n\nRule LC1: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

    }
};

rule LanguageCompatible2
{
   priority = high;
   when
    {
         ?c1: Component( lang.equals("JDK1.1"); ?i1:id);
         ?c2: Component( lang.equals("C++"); ?i2:id ; ?i1 != ?i2);

         ?e: Evaluation();

    }
   then
    {

         modify ?e {score += 1;}
```

```
            System.out.print("\n\nRule LC2: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

     }
};


rule LanguageCompatible3
{
   priority = high;
   when
     {
            ?c1: Component( lang.equals("JDK1.1"); ?i1:id);
            ?c2: Component( lang.equals("C"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

     }
   then
     {

            modify ?e {score += 6;}

            System.out.print("\n\nRule LC3: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

     }
};



rule LanguageCompatible4
{
   priority = high;
   when
     {
            ?c1: Component( lang.equals("JDK1.2"); ?i1:id);
            ?c2: Component( lang.equals("JDK1.2"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

     }
   then
     {

            modify ?e {score += 10;}

            System.out.print("\n\nRule LC4: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

     }
};
```

```
rule LanguageCompatible5
{
   priority = high;
   when
     {
          ?c1: Component( lang.equals("JDK1.1"); ?i1:id);
          ?c2: Component( lang.equals("JDK1.1"); ?i2:id ; ?i1 > ?i2);


          ?e: Evaluation();

     }
   then
     {

          modify ?e {score += 10;}

          System.out.print("\n\nRule LC5: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

     }
};


rule LanguageCompatible6
{
   priority = high;
   when
     {
          ?c1: Component( lang.equals("JDK1.0"); ?i1:id);
          ?c2: Component( lang.equals("JDK1.0"); ?i2:id ; ?i1 != ?i2);

          ?e: Evaluation();

     }
   then
     {

          modify ?e {score += 10;}

          System.out.print("\n\nRule LC6: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

     }
};


rule LanguageCompatible7
{
   priority = high;
   when
     {
          ?c1: Component( lang.equals("JDK1.0"); ?i1:id);
          ?c2: Component( lang.equals("JDK1.2"); ?i2:id ; ?i1 != ?i2);
```

```
            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 7;}

            System.out.print("\n\nRule LC7: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

      }
};



rule LanguageCompatible8
{
    priority = high;
    when
      {
            ?c1: Component( lang.equals("JDK1.0"); ?i1:id);
            ?c2: Component( lang.equals("JDK1.1"); ?i2:id ; ?i1 != ?i2);

            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 8;}

            System.out.print("\n\nRule LC8: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

      }
};

rule LanguageCompatible9
{
    priority = high;
    when
      {
            ?c1: Component( lang.equals("JDK1.2"); ?i1:id);
            ?c2: Component( lang.equals("C"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 6;}
```

```
            System.out.print("\n\nRule LC9: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

     }
};

rule LanguageCompatible10
{
   priority = high;
   when
     {
            ?c1: Component( lang.equals("JDK1.0"); ?i1:id);
            ?c2: Component( lang.equals("C"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

     }
   then
     {

            modify ?e {score -= 10;}

            System.out.print("\n\nRule LC10: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

     }
};


rule LanguageCompatible11
{
   priority = high;
   when
     {
            ?c1: Component( lang.equals("JDK1.2"); ?i1:id);
            ?c2: Component( lang.equals("C++"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

     }
   then
     {

            modify ?e {score += 1;}

            System.out.print("\n\nRule LC11: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

     }
};


rule LanguageCompatible12
{
```

```
    priority = high;
    when
      {
            ?c1: Component( lang.equals("JDK1.0"); ?i1:id);
            ?c2: Component( lang.equals("C++"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {

            modify ?e {score -= 10;}

            System.out.print("\n\nRule LC12: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};



rule PlatformCompatible1 {
    priority = high;
    when
      {
            ?c1: Component( platform.equals("Solaris"); ?i1:id);
            ?c2: Component( platform.equals("Java"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PC1: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

      }
};


rule PlatformCompatible2 {
    priority = high;
    when
      {
            ?c1: Component( platform.equals("MS Windows"); ?i1:id);
            ?c2: Component( platform.equals("Java"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
```

```
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PC2: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

      }
};

rule PlatformCompatible3 {
    priority = high;
    when
      {
            ?c1: Component( platform.equals("Solaris libraries"); ?i1:id);
            ?c2: Component( platform.equals("Solaris C Code"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {


            modify ?e {score += 10;}

            System.out.print("\n\nRule PC3: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

      }
};


rule PlatformCompatible4 {
    priority = high;
    when
      {
            ?c1: Component( platform.equals("Windows DLL"); ?i1:id);
            ?c2: Component( platform.equals("Windows binaries"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PC4: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

      }
};
```

```
rule PlatformCompatible5 {
   priority = high;
   when
    {
          ?c1: Component( platform.equals("Windows DLL"); ?i1:id);
          ?c2: Component( platform.equals("Solaris"); ?i2:id ; ?i1 != ?i2);


          ?e: Evaluation();

    }
   then
    {

          modify ?e {score -= 20;}

          System.out.print("\n\nRule PC5: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

    }
};


rule PlatformCompatible6 {
   priority = high;
   when
    {
          ?c1: Component( platform.equals("MS Windows"); ?i1:id);
          ?c2: Component( platform.equals("Solaris libraries"); ?i2:id ; ?i1 != ?i2);


          ?e: Evaluation();

    }
   then
    {

          modify ?e {score -= 20;}

          System.out.print("\n\nRule PC6: Component " + ?i1 + " and component " + ?i2 + ":
new score " + e.score + "...\n");

    }
};


rule ProtocolCompatible1 {
   priority = high;
   when
    {
          ?c1: Component( protocol.equals("RMI1.1"); ?i1:id);
          ?c2: Component( protocol.equals("RMI1.0"); ?i2:id ; ?i1 != ?i2);


          ?e: Evaluation();

    }
```

```
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PrC1: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};


rule ProtocolCompatible2 {
    priority = high;
    when
      {
            ?c1: Component( protocol.equals("RMI1.0"); ?i1:id);
            ?c2: Component( protocol.equals("RMI1.0"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PrC2: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};


rule ProtocolCompatible3 {
    priority = high;
    when
      {
            ?c1: Component( protocol.equals("RMI1.1"); ?i1:id);
            ?c2: Component( protocol.equals("RMI1.1"); ?i2:id ; ?i1 != ?i2);

            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PrC3: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};

rule ProtocolCompatible4 {
```

```
    priority = high;
    when
      {
            ?c1: Component( protocol.equals("RMI1.2"); ?i1:id);
            ?c2: Component( protocol.equals("RMI1.2"); ?i2:id ; ?i1 != ?i2);

            ?e: Evaluation();

      }
    then
      {

            modify ?e {score += 10;}

            System.out.print("\n\nRule PrC4: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};


rule ProtocolCompatible5 {
    priority = high;
    when
      {
            ?c1: Component( protocol.equals("RMI1.2"); ?i1:id);
            ?c2: Component( protocol.equals("RMI1.1"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {

            modify ?e {score -= 10;}

            System.out.print("\n\nRule PrC5: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};

rule ProtocolCompatible6 {
    priority = high;
    when
      {
            ?c1: Component( protocol.equals("RMI1.2"); ?i1:id);
            ?c2: Component( protocol.equals("RMI1.0"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

      }
    then
      {
```

```
            modify ?e {score -= 10;}

            System.out.print("\n\nRule PrC6: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

     }
};


rule GUICompatible1 {
   priority = high;
   when
     {
            ?c1: Component( gui.equals("AWT"); ?i1:id);
            ?c2: Component( gui.equals("AWT"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

     }
   then
     {

            modify ?e {score += 10;}

            System.out.print("\n\nRule GC1: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

     }
};


rule GUICompatible2 {
   priority = high;
   when
     {
            ?c1: Component( gui.equals("Swing"); ?i1:id);
            ?c2: Component( gui.equals("Swing"); ?i2:id ; ?i1 != ?i2);


            ?e: Evaluation();

     }
   then
     {

            modify ?e {score += 10;}

            System.out.print("\n\nRule GC2: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

     }
};

rule GUICompatible3 {
   priority = high;
```

```
    when
      {
            ?c1: Component( gui.equals("Swing"); ?i1:id);
            ?c2: Component( gui.equals("AWT"); ?i2:id ; ?i1 != ?i2);

            ?e: Evaluation();

      }
    then
      {

            modify ?e {score -= 5;}

            System.out.print("\n\nRule GC3: Component " + ?i1 + " and component " + ?i2 +
": new score " + e.score + "...\n");

      }
};
```

APPENDIX C

Java programs that perform comparison between components in each ensemble and integration rules in the database compose of 3 programs: Main.java, Component.java, and Evaluation.java as shown below.

```
// Main.java
// The main function for the default version of the Component Analyzer

package component;

import java.io.*;
import ilog.rules.engine.*;

public class Main
{
   private static void waitReturnKey()
    {
          String stop = System.getProperty("IlrWaitReturnKey");
          if (stop != null) {
             System.out.println("Press return to continue . . .");
             try {
                   System.in.read();
             } catch (IOException e) {
                   // Does nothing.
             }
          }
    }

   public static void main(String[] args)
    {
          String filename = "data/ensemble.ilr";
          if (args.length > 0) filename = args[0];

          IlrRuleset ruleset = new IlrRuleset();
          boolean parsed = ruleset.parseFileName(filename);

          if (!parsed) return;

          IlrContext context = new IlrContext(ruleset);

          IlrRuntime.self.timeSnapshot();
          int nrules = context.fireAllRules();
          IlrRuntime.self.timeSnapshot();
        IlrRuntime.self.printTimeUsage(nrules);

          context.retractAll();
          context.end();
          waitReturnKey();
    }
};
```

// **Component.java**

```
package component;

public class Component
{
   public int id;
   public static String func, lang, platform, protocol, gui, framework;
   public int score[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
   public int f(int i) {
         return i+1; }

   public void show_scores() {
         int k;
         System.out.println("\nScores for component " + id + ":");
         for (k=0; k<= 8; k++) {
            System.out.println("Score[" + k + "] is " + score[k]);
         }

   }

   public Component()
     {
          func = "";
          lang = "";
          platform = "";
          protocol = "";
          gui = "";
          framework = "";

     }
};
```

// **Evaluation.java**

```
package component;

public class Evaluation
{

   public int score;

   public Evaluation()
     {
     }
};
```

APPENDIX D

The output from running the prototype is in the form of component identifiers of each ensemble, number of rules fired and compatibility score (new score) as shown below.

<component id='100'>

<component id='101'>

<component id='109'>

Rule PrC2: Component 101 and component 109: new score 10...

Rule PrC6: Component 100 and component 109: new score 0...

Rule GC3: Component 100 and component 101: new score -5...

Rule PrC6: Component 100 and component 101: new score -15...

4 rules fired in 0.009999999776482582 seconds.

400 rules per second.

<component id='100'>

<component id='103'>

<component id='107'>

Rule GC2: Component 100 and component 107: new score 10...

Rule LC4: Component 100 and component 107: new score 20...

Rule PrC3: Component 103 and component 107: new score 30...

Rule PrC5: Component 100 and component 107: new score 20...

Rule LC11: Component 100 and component 103: new score 21...

Rule PrC5: Component 100 and component 103: new score 11...

6 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='115'>

<component id='103'>

<component id='106'>

Rule GC2: Component 106 and component 115: new score 10...

Rule PrC3: Component 103 and component 106: new score 20...

2 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='100'>

<component id='101'>

<component id='105'>

Rule GC2: Component 100 and component 105: new score 10...

Rule LC1: Component 101 and component 105: new score 19...

Rule LC4: Component 100 and component 105: new score 29...

Rule PC6: Component 100 and component 105: new score 39...

Rule PrC5: Component 100 and component 105: new score 29...

Rule GC3: Component 100 and component 101: new score 24...

Rule PrC6: Component 100 and component 101: new score 14...

7 rules fired in 0.009999999776482582 seconds.

700 rules per second.

<component id='100'>

<component id='103'>

<component id='1001'>

Rule PrC4: Component 100 and component 1001: new score 10...

Rule LC11: Component 100 and component 103: new score 11...

Rule PrC5: Component 100 and component 103: new score 1...

3 rules fired in 0.009999999776482582 seconds.

300 rules per second.

<component id='100'>

<component id='101'>

<component id='106'>

Rule GC2: Component 100 and component 106: new score 10...

Rule LC1: Component 101 and component 106: new score 19...

Rule LC4: Component 100 and component 106: new score 29...

Rule PC6: Component 100 and component 106: new score 36...

Rule PrC5: Component 100 and component 106: new score 26...

Rule GC3: Component 100 and component 101: new score 21...

Rule PrC6: Component 100 and component 101: new score 11...

7 rules fired in 0.009999999776482582 seconds.

700 rules per second.

<component id='210'>

<component id='103'>

<component id='1001'>

Rule PrC1: Component 103 and component 210: new score 10...

1 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='210'>

<component id='101'>

<component id='106'>

Rule GC2: Component 106 and component 210: new score 10...

Rule LC1: Component 101 and component 106: new score 19...

Rule PrC1: Component 106 and component 210: new score 29...

Rule PC5: Component 101 and component 210: new score 19...

Rule PrC2: Component 101 and component 210: new score 29...

5 rules fired in 0.009999999776482582 seconds.

500 rules per second.

<component id='100'>

<component id='101'>

<component id='1001'>

Rule PrC4: Component 100 and component 1001: new score 10...

Rule GC3: Component 100 and component 101: new score 5...

Rule PrC6: Component 100 and component 101: new score -5...

3 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='115'>

<component id='101'>

<component id='1001'>

Rule PrC4: Component 115 and component 1001: new score 10...

Rule LC5: Component 115 and component 101: new score 20...

2 rules fired in 0.009999999776482582 seconds.

200 rules per second.

<component id='100'>

<component id='103'>

<component id='106'>

Rule GC2: Component 100 and component 106: new score 10...

Rule LC4: Component 100 and component 106: new score 20...

Rule PC6: Component 100 and component 106: new score 27...

Rule PrC3: Component 103 and component 106: new score 37...

Rule PrC5: Component 100 and component 106: new score 27...

Rule LC11: Component 100 and component 103: new score 28...

Rule PrC5: Component 100 and component 103: new score 18...

7 rules fired in 0.020999999716877937 seconds.

333 rules per second.

<component id='115'>

<component id='101'>

<component id='1001'>

Rule PrC4: Component 115 and component 1001: new score 10...

Rule LC5: Component 115 and component 101: new score 20...

2 rules fired in 0.009999999776482582 seconds.

200 rules per second.

<component id='100'>

<component id='101'>

<component id='1001'>

Rule PrC4: Component 100 and component 1001: new score 10...

Rule GC3: Component 100 and component 101: new score 5...

Rule PrC6: Component 100 and component 101: new score -5...

3 rules fired in 0.009999999776482582 seconds.

300 rules per second.

<component id='100'>

<component id='103'>

<component id='1001'>

Rule PrC4: Component 100 and component 1001: new score 10...

Rule LC11: Component 100 and component 103: new score 11...

Rule PrC5: Component 100 and component 103: new score 1...

3 rules fired in 0.009999999776482582 seconds.

300 rules per second.

<component id='210'>

<component id='103'>

<component id='1001'>

Rule PrC1: Component 103 and component 210: new score 10...

1 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='100'>

<component id='103'>

<component id='107'>

Rule GC2: Component 100 and component 107: new score 10...

Rule LC4: Component 100 and component 107: new score 20...

Rule PrC3: Component 103 and component 107: new score 30...

Rule PrC5: Component 100 and component 107: new score 20...

Rule LC11: Component 100 and component 103: new score 21...

Rule PrC5: Component 100 and component 103: new score 11...

6 rules fired in 0.009999999776482582 seconds.

600 rules per second.

<component id='210'>

<component id='101'>

<component id='105'>

Rule GC2: Component 105 and component 210: new score 10...

Rule LC1: Component 101 and component 105: new score 19...

Rule PrC1: Component 105 and component 210: new score 29...

Rule PC5: Component 101 and component 210: new score 19...

Rule PrC2: Component 101 and component 210: new score 29...

5 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='100'>

<component id='101'>

<component id='1001'>

Rule PrC4: Component 100 and component 1001: new score 10...

Rule GC3: Component 100 and component 101: new score 5...

Rule PrC6: Component 100 and component 101: new score -5...

3 rules fired in 0.019999999552965164 seconds.

150 rules per second.

<component id='100'>

<component id='103'>

<component id='107'>

Rule GC2: Component 100 and component 107: new score 10...

Rule LC4: Component 100 and component 107: new score 20...

Rule PrC3: Component 103 and component 107: new score 30...

Rule PrC5: Component 100 and component 107: new score 20...

Rule LC11: Component 100 and component 103: new score 21...

Rule PrC5: Component 100 and component 103: new score 11...

6 rules fired in 0.019999999552965164 seconds.

300 rules per second.

<component id='210'>

<component id='101'>

<component id='107'>

Rule GC2: Component 107 and component 210: new score 10...

Rule LC1: Component 101 and component 107: new score 19...

Rule PC4: Component 101 and component 107: new score 29...

Rule PrC1: Component 107 and component 210: new score 39...

Rule PC5: Component 101 and component 210: new score 29...

Rule PrC2: Component 101 and component 210: new score 39...

6 rules fired in 0.010999999940395355 seconds.

545 rules per second.

<component id='115'>

<component id='101'>

<component id='107'>

Rule GC2: Component 107 and component 115: new score 10...

Rule LC1: Component 101 and component 107: new score 19...

Rule PC4: Component 101 and component 107: new score 29...

Rule LC5: Component 115 and component 101: new score 39...

4 rules fired in 0.009999999776482582 seconds.

400 rules per second.

<component id='100'>

<component id='101'>

<component id='109'>

Rule PrC2: Component 101 and component 109: new score 10...

Rule PrC6: Component 100 and component 109: new score 0...

Rule GC3: Component 100 and component 101: new score -5...

Rule PrC6: Component 100 and component 101: new score -15...

4 rules fired in 0.009999999776482582 seconds.

400 rules per second.

<component id='100'>

<component id='101'>

<component id='1001'>

Rule PrC4: Component 100 and component 1001: new score 10...

Rule GC3: Component 100 and component 101: new score 5...

Rule PrC6: Component 100 and component 101: new score -5...

3 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='115'>

<component id='101'>

<component id='105'>

Rule GC2: Component 105 and component 115: new score 10...

Rule LC1: Component 101 and component 105: new score 19...

Rule LC5: Component 115 and component 101: new score 29...

3 rules fired in 0.0 seconds.

9223372036854775807 rules per second.

<component id='100'>

<component id='101'>

<component id='105'>

Rule GC2: Component 100 and component 105: new score 10...

Rule LC1: Component 101 and component 105: new score 19...

Rule LC4: Component 100 and component 105: new score 29...

Rule PC6: Component 100 and component 105: new score 39...

Rule PrC5: Component 100 and component 105: new score 29...

Rule GC3: Component 100 and component 101: new score 24...

Rule PrC6: Component 100 and component 101: new score 14...

7 rules fired in 0.009999999776482582 seconds.

700 rules per second.

<component id='210'>

<component id='101'>

<component id='105'>

Rule GC2: Component 105 and component 210: new score 10...

Rule LC1: Component 101 and component 105: new score 19...

Rule PrC1: Component 105 and component 210: new score 29...

Rule PC5: Component 101 and component 210: new score 19...

Rule PrC2: Component 101 and component 210: new score 29...

5 rules fired in 0.009999999776482582 seconds.

500 rules per second.

APPENDIX E

Chief Editor

26 February 2002

Ms Somjai Boonsiri
Department of Mathematics
Faculty of Science
Chulalongkom University
Phayathai Road
Bangkok Thailand 10330

Dear Ms Somjai

PAPER TITLED "AUTOMATED COMPONENT ENSEMBLE EVALUATION"

Please be informed that your paper has been reviewed and accepted. It will be included in the next volume of our online journal.

Thank you for submitting your paper for International Journal of Information Technology (IJIT). We look forward to your continued support as we strive for greater excellence.

With best regards

Yours sincerely

Jane Chan
for Prof Robert Gay
Chief Editor, IJIT

SINGAPORE COMPUTER SOCIETY
53/63A NEIL ROAD
SINGAPORE 088891
TEL: 2262-567 FAX. 2262-969

BIOGRAPHY

Name – Last Name : Somjai Boonsiri

Current Profession : Assistant Professor at Department of Mathematics, Faculty of Science, Chulalongkorn University.

Education : M. Sc. Computer Science, Chulalongkorn University (1992)

: B. Sc. Mathematics, Chiang Mai University (1984)

Research Areas : Component-based software engineering, Software reuse, Distributed system, Database systems

Training : 1. Researched at the Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, U.S.A. 1 year.

2. UNIX-LAN System Design and Development training at Center of the International Cooperation for Computerization (CICC), Japan 2 months.

Experiences : Lecturer at Payap University (1986-1989)

: Lecturer at Chulalongkorn University (1993-1995)

Publications : 1. Boonsiri, S. and Pattarakosol, P.,"Starting to Computer" (in Thai), AR Information & Publication, 1997.

2. Boonsiri, S. et al., "Internet: Variety Services" (in Thai), S.D.Press Ltd., 1996.

3. Robert C. Seacord, David Mundie, and Somjai Boonsiri, "K-BACEE: Knowledge-Based Automated Component Ensemble Evaluation", Proceedings of the 27th IEEE EUROMICRO Conference, September 2001.

4. Somjai Boonsiri, Robert C. Seacord, and Russ Bunting, "Automated Component Ensemble Evaluation", to be published in the International Journal of Information Technology (IJIT) in volume 8.