A Repository Framework for Self-Growing Robot Software

Hyung-Min Koo, In-Young Ko

Information and Communications University (ICU)

119 Munjiro, Yuseong-gu, Daejeon, 305-732, Korea

{hyungminkoo, iko}@icu.ac.kr

Abstract

Self-growing software is a software system that grows its functionalities and configurations by itself based on dynamically monitored situations. Self-growing software is especially necessary for intelligent service robots, which monitor their surrounding environments and provide appropriate behaviors for human users. Intelligent service robots often face problems that cannot be resolved with the conventional software technology. To support self-growing software for intelligent service robots, the SemBots project at ICU is developing a repository framework that allows robot software to dynamically acquire software components that are necessary to resolve a dynamic situation. In this paper, we describe the requirements and architecture of the repository system for self-growing software. We also present a prototype implementation of the repository system.

Keywords *: Self-growing Software, Self-adaptive Software Intelligent Service Robots, Component Repositories*

1. Introduction

As the complexity and dynamism of computing grow rapidly, people seek for self-adaptive software systems that change their behaviors dynamically based on changing environments and requirements [1]. Self-growing software is a kind of self-adaptive software. Self-growing is an action of improving the software system through self-evolution and self-expansion, without human intervention. A similar concept is discussed in the autonomic computing that defines self-configuration, self-optimization, self-healing, and self-protection as the essential functions of self-managed software [2]. In [2], self-optimization is defined as "an autonomic system that continually seeks ways to improve its operation, identifying and seizing opportunities to make itself more efficient in aspect of performance or cost" [2]. Therefore, self-growing is tightly related to the self-optimization issue in autonomic computing.

Self-growing software is especially necessary for intelligent service robots, which monitor their surrounding environments and provide appropriate behaviors for human users. Intelligent service robots often face problems that cannot be resolved with the conventional software technology.

To support self-growing software for intelligent service robots, the SemBots project at ICU is developing a repository framework that allows robot software to dynamically acquire software components that are necessary to resolve a dynamic situation. In this paper, we describe the requirements and architecture of the repository system for self-growing software. We also present a prototype implementation of the repository system.

In section 2, we explain what self-growing software is, and the differences between self-growing software and typical software systems. We also describe what processes are needed to support self-growing software. In Section 3, we briefly introduce the SemBots project that is currently developing self-growing software technologies. In Section 4, we explain requirements of the repository system for self-growing software. In Sections 5 and 6, we explain our approaches and the prototype implementation of the repository system respectively. We evaluate our repository system in Section 7, and discuss about the related work in Section 8. Finally, we conclude the paper in Section 9.

2. Self-Growing Software

Self-growing software is a software system that grows its functionalities and configurations by itself based on dynamically monitored situations. A self-growing software system proactively searches for external modules (objects, components, and services) when it faces a situation in which the current software cannot support functionalities that are necessary to cope with a problem. By using self-growing software, we can improve the efficiency of software system dramatically, especially for the software systems that require working in a dynamically changing environment.

2.1 Self-Growing Software vs. Typical Software

In a typical software system, to make software have a new set of functionalities, we need to shut down the system and reconfigure the software with a new set of software components. As depicted in Fig.1, during the system development (or redesign) time, users' requirements need to be explicitly considered by a system administrator to select a new set of components that are necessary to reconfigure the software system.

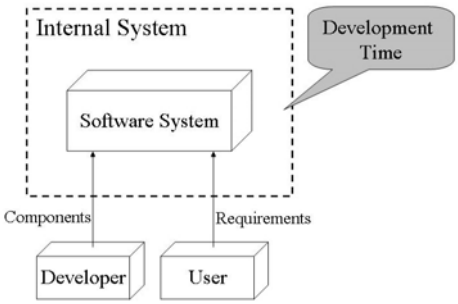

Fig. 1 A Typical Software System

In contrast to the typical software system, in a self-growing software system, the software can 'grow' at runtime. As illustrated in Fig. 2, developers develop components and register them into an external environment. Self-growing software acquires these components dynamically during runtime if a problem situation is detected or user requirements are changed.

2.2 The Process of Self-Growing

To accomplish 'self-growing', three steps are necessary: collecting, learning and growing.

i) *Collecting*: Contacting external component repositories to acquire new components that are needed to enhance the functionality of the software system to cope with unexpected changes.

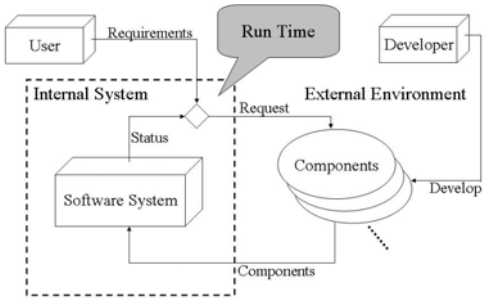

Fig. 2 A Self-Growing Software System

ii) *Learning*: Considering the history of selecting components for a certain situation to select the most appropriate components among the collected components

iii) *Growing*: Storing the selected components into the internal repository

As shown in Fig. 3, these three steps are repeated continually, and software grows gradually.

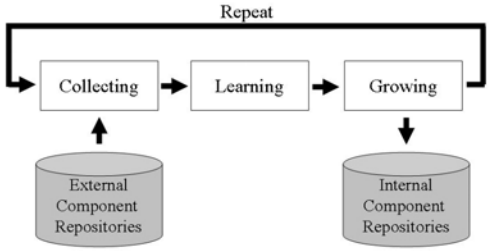

Fig. 3 The Process of Self-Growing

In this paper, we focus on the repository structure for intelligent service robot software. Especially, the rest of the paper focuses on explaining about the overall system architecture, and detail mechanism to realize the collecting and growing steps.

3. The SemBots Project

The SemBots project at ICU is developing a software framework to support self-growing software. This software framework allows robots to deal with unexpected situations by improving the mechanism of solving problems and by supporting a high level software configuration mechanism. As Fig. 4 explains, when

environmental changes, user requirements changes or exceptional situations are encountered, the robot software grows itself by gathering relevant components from external software component repositories, and by reconfiguring the software system to utilize the new components.

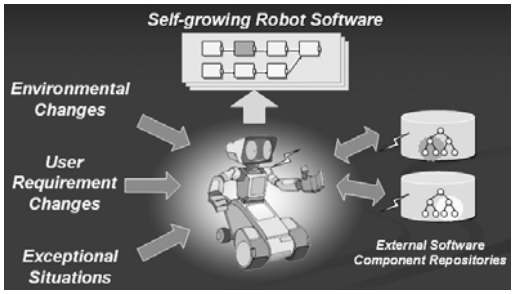

Fig. 4 Self-Growing Software

In the SemBots project, we are also developing a component broker for intelligent service robots. The component broker provides facilities that allow a robot to choose the most appropriate and effective components for a situation. An ontology-based semantic representation model [3] and a semantic gauging mechanism are the essential elements of the component broker.

4. Requirements of the Repository System for Self-Growing Software

4.1 Accuracy

To support the self-growing capability, component repositories must maintain accurate metadata about the software components that they manage. The repository system must also provide a mechanism to use the metadata in searching components that are appropriate to solve a problematic situation encountered.

4.2 Scalability

The repository system must be scalable to seamlessly utilize various software componentst that are scattered across external network resources. The efforts of incorporating additional external resources must be minimized.

4.3 Dynamism

Runtime software reconfiguration is one of the key elements of self-growing software. Therefore, the component repository system must support functions for dynamically collecting and storing software components. These activities also have to be done quickly enough so that the system's performance is not degraded significantly by the actions.

4.4 Accessibility

The repository system must provide a way to transparently access information about software components that are distributed across internal and external component repositories.

4.5 Changeability

The capacity of an internal repository is limited. Therefore, it is necessary have criteria for changing information stored in internal repositories. These criteria are used to decide what information has to be retired from an internal repository, and what information has to be added to an internal repository.

5. The Repository Architecture

In this section, we describe our approaches for the repository framework for self-growing robot software. We also explain the main architecture of the repository framework.

5.1 Approaches

· **Semantic Matching:** To collect components accurately, we use an ontology-based semantic matching mechanism. When a robot contacts an external repository to search for a component, it firstly collects ontologies of candidate components from external ontology repositories. Based on the ontologies collected, the component broker identifies the most appropriate component for a situation.

· **Ontology Repositories:** An ontology repository stores ontology-based descriptions of software components. In an ontology repository, component ontologies are stored in the form of a graph that represents functionality, input/output semantics, and other conditions of components and relationships among ontologies. The component broker uses the ontology repository to search for appropriate components.

· **Component Repositories:** The component repository stores component URIs (Uniform Resource Identifiers) and physical component files. A component instance description in an ontology repository points to a physical

component file in a component repository.

· **A Component Acquisition Engine:** A component acquisition engine provides functions to collect component ontologies and component files from external ontology and component repositories. Based on a query sent by the component broker, the component acquisition engine checks if the required component is available in an internal repository. If there is no component matched in an internal repository, the component acquisition engine identifies and locates an appropriate external repository to access. As explained earlier, ontologies of candidate components are firstly collected from external ontology repositories, and then the component broker decides a specific component instance to use. Only the component selected by the component broker is physically accessed from an external repository and stored into an internal repository. This is an important factor to enable high-performance component acquisition.

· **Component Retirement Plan:** The self-growing behavior can not be performed infinitely because internal resources are limited. Therefore, a special plan for retiring and alternating components is essential for internal repositories. In our repository framework, each component description includes an access counter and has a field to record the acquisition date. When a robot faces resource limitation, it searches the most unused components and retires them from internal repositories. Older ones among the most unused component are retired first. Critical components that are essential for robot behaviors are not considered as retirement candidates.

5.2 The Overall Repository Architecture

Fig. 5 describes the overall repository architecture. When an unexpected situation is occurred, the component broker firstly searches the internal ontology repository. If the component broker cannot find any component ontologies that are appropriate for the situation, it requests the component acquisition engine to contact an external component repository. The component acquisition engine searches for proper external component ontologies, and uploads them into the internal ontology repository. After loading the component descriptions, based on the decision made by the component broker, the component acquisition engine retrieves a specific component file from an external component repository and store it into the internal component repository.

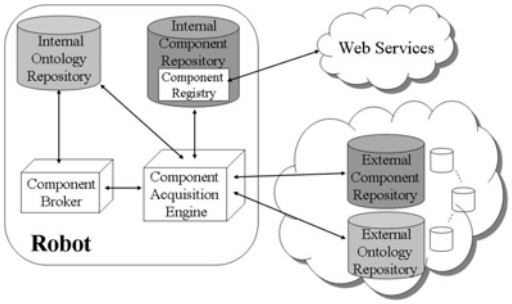

Fig. 5 The Repository Architecture

5.3 Accessing Internal Repositories

By accessing internal repositories, the component broker searches for the components that provide functionalities to handle a situation detected. It extracts a set of component candidates by using a semantically-based interoperability measurement [20], and sends them to a learning engine. The leaning engine chooses the most appropriate component based on the history of utilizing components for a certain set of situations. Fig. 6 explains the interactions among repository elements in accessing internal repositories.

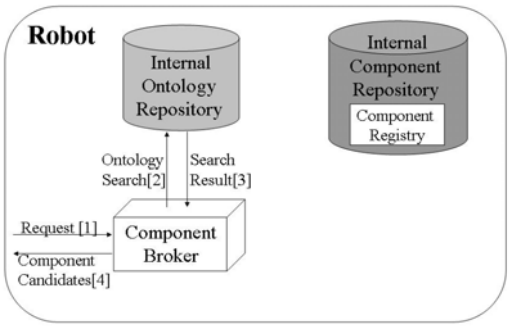

Fig. 6 Accessing Internal Repositories

5.4 Accessing External Repositories

If there is no available or suitable component in the internal repository, the component broker sends a request to the component acquisition engine (see Fig. 7). The component acquisition engine then collects component ontologies from various external ontology repositories, and the component broker measures the semantic interoperability of them with the existing components in the software system. After the

learning engine decides the candidate components to use, the component acquisition engine uploads a set of ontology-based descriptions of the components into the internal ontology repository and acquires the physical component files from external component repositories. All the acquired components are stored into the internal component repository.

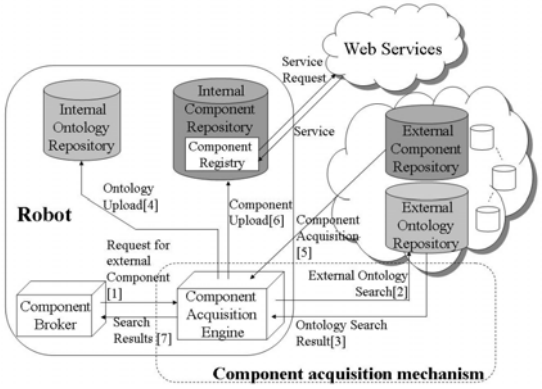

Fig. 7 Accessing External Repositories

5.5 Accessing Web Services

Robots sometimes need functionalities that are not supported as software components. For instance, if a robot's behavior depends on weather, the robot needs to access weather information from the Web to accomplish its task. To access weather information, the robot can access a Web service that provides weather-related services.

If there is no available service in the local service registry, the robot accesses Web service repositories, UDDI (Universal Description, Discovery and Integration) registries to find appropriate Web services and sends requests to the services (see Fig. 8). The robot stores the retrieved service information into the local component registry so that the local registry can grow with a new set of available services.

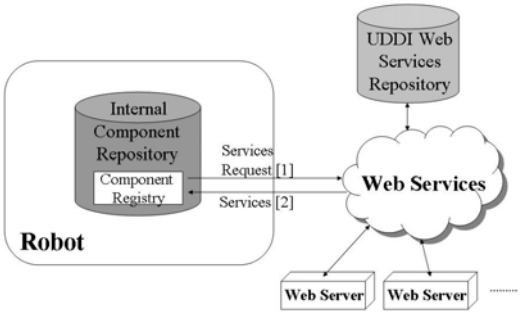

Fig. 8 Accessing Web Services

6. A Prototype

In this section, we describe a prototype implementation of the repository system. Fig. 9 shows the conceptual architecture of the prototype system. This architecture is being developed in our project.

The *monitor* infers robot's situation based on data received from sensors (Laser sensors, infra-red sensors, and vision sensors are available in the current robot platform). The *component broker* finds a strategy to solve the inferred situation, and searches for candidate components that are necessary to perform the strategy.

This set of candidates is then transferred to the *learning engine* to decide the most appropriate components based on past experiences. Finally, the selected component is used for reconfiguring the robot software.

If there is no available component for solving the situation, the component broker requests the *component acquisition engine* for external components. The acquisition engine collects ontologies from external ontology repositories, updates them into internal ontology repositories, and responses to the component broker. The component broker decides an external component to use with the help of the learning engine.. The acquisition engine retrieves the actual component file from an external component repository and stores it into the internal component repository. The acquired component is then used for reconfiguraing the robot software.

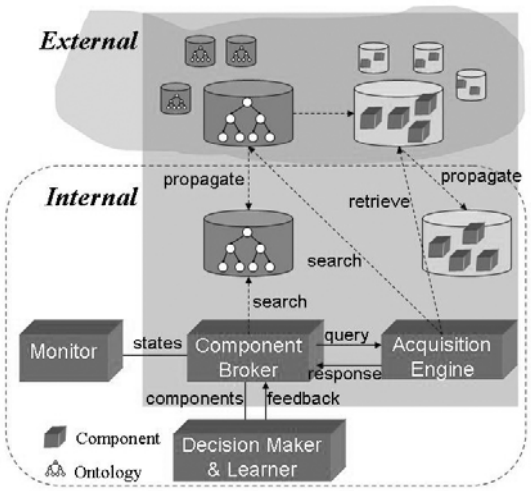

Fig. 9 The Conceptual Architecture of the Prototype System

We are currently implementing the repository system for self-growing robot software on the Microsoft Windows XP operating system. We use the JAVA programming language for implementing the system. For implementing the repositories, we use the Protégé tool, Jena2 library, My-SQL, JDBC, RMI, and FTP. We use Protégé to make RDF and RDFS files [4]. These files contain ontologies information in the form of XML files. Fig. 10 shows ontologies for intelligent service robots [21].

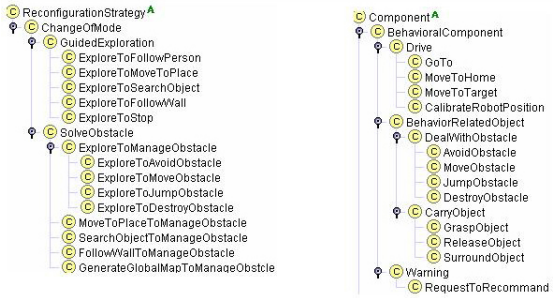

Fig. 10 Ontologies for Intelligent Service Robots (Edited with Protégé)

Jena2 provides a set of methods for querying ontologies from RDF and RDFS files. The Jena2 library provides two ways of handling ontology models: memory models, and database models. It provides the “modelRDB” class to store ontology models into a database. By using the “modelRDB” class, we can store ontology hierarchy into a database in forms of tables [5].

Fig. 11 depicts the process of creating the initial ontology repository, making RDF and RDFS files with Protégé, integrating the file data with Jena2, and storing them into a database.

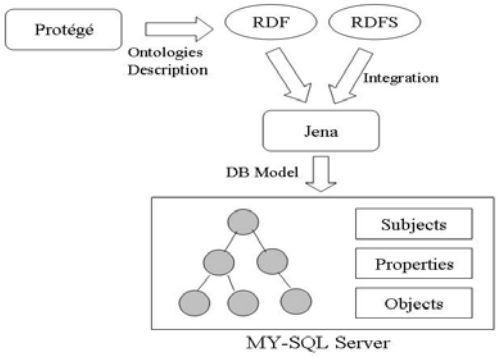

Fig. 11 The Process of Creating an Ontology Repository

We used My-SQL to build databases for ontology and component repositories. An ontology repository includes a RDF triple model [5] of ontologies, and a component repository contains URIs of components and locations of component files. To connect Jena with the database, we used a JDBC driver. RMI is used for accessing external repositories for collecting external ontologies. The component acquisition engine makes a remote procedure call to the remote method that is running on an external repository server. After accessing component ontologies by using RMI, the component acquisition engine uses FTP to actually download external components.

Fig. 12 shows an implemented architecture of the repository system. The RMI handler calls a remote method with a component schema for collecting external ontologies from external ontology repository. This RMI handler supports the scalability and accessibility requirements.

The *external ontology manager* searches for proper component ontologies in external ontology repositories and measures semantic interoperability among them. It then returns candidates components to the component acquisition engine in the internal system. This manager satisfies accuracy requirement.

By searching proper locations of the candidate components, the *ontology updater* uploads the component instance descriptions into the internal ontology repository. Based on the feedback from the learning engine, the most appropriate component is selected.

Finally, the component receiver accesses the FTP server of the external component repository, retrieves the component file by referring to the component’s location, and uploads the component file into the internal component repository. The activities of updating ontologies by the *ontology updater*, and uploading component files by the *component receiver* are performed during run time. This satisfies the dynamism requirement.

When internal repositories meet their capacity limitations, the *retirement planner* searches for the most unused and/or oldest components to be retired. After the decision, the retirement planner retires them from internal repositories. This planner supports the changeability requirement.

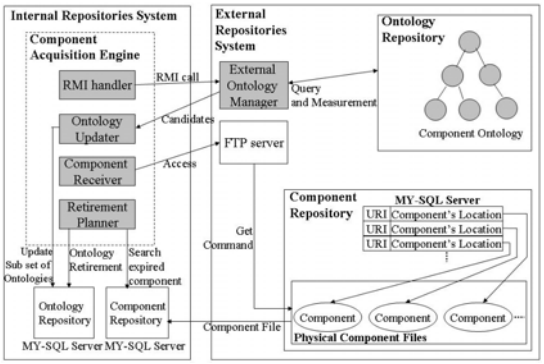

Fig. 12 The Implemented Architecture

7. Evaluation

· **Accuracy**: We use the semantic matching mechanism. We develop ontology-based component repositories, and this makes the component broker collect enough and accurate information about components. Therefore, our framework provides a mechanism for supporting 'accuracy'.

· **Scalability**: We divide the repositories into internal and external repositories. Based on this repository structure, a robot can utilize many external repositories and external computing resources. By making the internal repositories small and efficient, and by making it extensible by accessing external repositories, we achieve the scalability and extensibility requirements.

· **Dynamism**: The self-growing process is done during runtime, and components can be collected and utilized dynamically. In addition, the updates of ontologies and components are performed during runtime.

· **Accessibility**: In addition to the storage space, external repositories provide the functions to process queries and to compare between components. Robots can collect external ontologies by calling the remote method in external repositories. This repository architecture makes robots access various external repositories easily and efficiently.

· **Changeability**: When a robot meets the limitation of resources, it actuates the component retirement process. We use two kinds of retirement plan as explained earlier. This retirement plan satisfies the changeability requirement.

8. Related Work

There have been several researches about component repositories and self-growing software. Most of the repositories use keyword-based matching, indexing and browsing mechanisms [13].

· **Ontology-based Repositories**: There are some researches of ontology-based repositories. Semantic Web languages such as RDF and OWL can represent metadata of information, and can deal with ontologies and their repository [8,18]. To access distributed ontology repositories, APIs such as OKBC [19] are also available.

· **Component Repositories**: There are many researches about component repositories to provide component retrieval: CRPS [9], universal repository [14], some commercial repositories [17], knowledge-based repository [12], repositories to support reusability [6, 7, 16], and active and effective repositories [10, 11, 15]. However, they mostly focus on description, reusability, publication, and reliability of components, and do not consider mechanisms for self-growing.

· **Self-Growing Software**: In the viewpoint of autonomic computing [2], self-growing can be characterized as a high performance and low cost optimization method.

9. Conclusions and Future Work

Self-growing software is important to support self-adaptiveness and autonomic computing. To realize self-growing software, the repository technology does an important role. We described a repository framework for self-growing robot software. To support self-growing, we provide the semantically-based component brokering mechanism, ontology repositories, component repositories, component acquisition engine, and component retirement plan. Our framework meets the requirements of accuracy, scalability, dynamism, accessibility, and changeability that are essential properties of software for intelligent service robots.

We are currently working on optimizing the

process of acquiring external software components and improving the overall performance of the repository system. In addition, we are currently applying our framework to a silver-mate robot platform. We are also working on utilizing Web services in composing and reconfiguring robot software.

10. Acknowledgement

This research was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Commerce, Industry and Energy of Korea.

The authors also would like to thank Hyun-il Shin, Yu-sik Park, Beom-jun Jeon, and Ki-hyeon Kim for their comments and insights.

References

[1] Peyman Oreizy and et al., “An Architecture-Based Approach to Self-Adaptive Software”, IEEE Intelligent System, 1999.

[2] Jeffrey O. Kephart and et al., “Vision of Autonomic Computing”, IEEE Computer Society, 2003.

[3] Vijayan S. and et al., “A Semantic-Based Approach to Component Retrieval”, ACM, SIGMIS Database, Vol. 34. No. 3, 2003.

[4] What is Protégé?. http://protege.stanford.edu/overview

[5] Shelly Powers, “Practical RDF”, P.16-P.22, O’Reilly publication, July, 2003.

[6] Yunwen Ye et al., “An Active and Adaptive Reuse Repository System”, IEEE, 2001.

[7] Jose Luis Barros Justo et al., “A Repository to support Specifications Reuse”, IEEE, 1996.

[8] Regina M. M. Braga et al., “The Use of Mediation and Ontology Technologies for Software Component Information Retrieval”, ACM, 2001.

[9] Jung-eun Cha et al., “Design and Implementation of Component Repository for Supporting Component Based Development Process” Software Engineering Department of ETRI, IEEE, 2001.

[10] Heinrich Jasper, “Active Databases for Active Repositories”, IEEE, 1994.

[11] Scott Henninger, “Supporting the Construction and Evolution of Component Repositories”, Proceedings of ICSE, IEEE, 1996.

[12] Padmal Vitharana et al., “Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis”, IEEE transactions on Software Engineering, Vol. 29, NO. 7, July 2003.

[13] Luqi and Jiang Guo, “Toward Automated Retrieval for Software Component Repository”, Dept. of computer science, Naval Postgraduate School.

[14] Sridhar Iyengar, “A Universal Repository Architecture using the OMG UML and MOF”, IEEE, 1998.

[15] Kurt Schneider et al., “Effective Experience Repositories for Software Engineering”, Proceeding of the 25th ICSE’ 03, IEEE, 2003.

[16] Jihyungg Lee et al., “Facilitating Reuse of Software Component using Repository Technology”, Proceeding of the Tenth Asia-Pacific Software Engineering Conference, IEEE, 2003.

[17] Luqi and Jiang Guo, “A Survey of Software Reuse Repository”, Research Associate US national Research Council.

[18] Natalya F. Noy and et al., “Making Biomedical Ontologies and Ontology Repositories Work”, IEEE Intelligent Systems, 2004.

[19] Richard Fikes and et al., “Distributed Repositories of High Expressive Reusable Ontologies”, IEEE Intelligent Systems, 1999.

[20] In-Young Ko, Robert Neches, and Ke-Thia Yao, “A Semantic Model and Composition Mechanism for Active Document Collection Templates in Web-based Information Management Systems”, Electronic Transactions on Artificial Intelligence (ETAI), Vol. 5, Section D, pp.55-77, 2001.

[21] Hwayoun Lee, Ho-Jin Choi, In-Young Ko. “A Semantically-Based Software Component Selection Mechanism for Intelligent Service Robots”, To appear in Proceedings of 4th Mexican International Conference on Artificial Intelligence (MICAI2005), Monterrey, Mexico, November 2005.