International Journal of Software Engineering and Knowledge Engineering

A FRAMEWORK FOR TOOL-BASED SOFTWARE ARCHITECTURE RECONSTRUCTION*

SUNGWON KANG †, SEONAH LEE ‡ and DANHYUNG LEE †

† *SOftware Technology Institute, Information and Communications University*
517-100, Dogok, Kangnam, Seoul, 135-120, Korea
{kangsw, danlee}@icu.ac.kr

‡ *Department of Computer Science, University of British Columbia*
201-2366 Main Mall, Vancouver, BC V6T 1Z4, Canada
salee@cs.ubc.ca

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

For software with nontrivial size and complexity, it is not feasible to manually perform architecture reconstruction. Therefore it is essential for the software architecture miner who is mining architecture from the existing software to have a well-defined software architecture reconstruction process that helps incorporate as much tool use as possible at the appropriate steps of architecture reconstruction. There are some existing software architecture reconstruction frameworks but they do not provide guidelines on how to systematically utilize tools to produce architecture views for a reconstruction purpose. In this paper, we propose a framework for tool-based software architecture reconstruction. This framework consists of a generic process for software architecture reconstruction and the steps to derive from it a concrete tool-based process to be used for actual architecture reconstruction. The architecture miner can use this framework to analyze source code for modifying source code as well as to reconstruct software architecture from source code.

Keywords: reverse engineering; software architecture, architecture reconstruction

1. Introduction

Architecture reconstruction is a reverse engineering activity that aims at recovering decisions that have either been lost or are unknown (Gorton et al., 2005). Since robust and clear software architecture is often the key determinant of the success or failure of many software projects (Gorton et al., 2005), software architecture reconstruction is important when the software architecture has been lost or eroded. Thus software

* This paper is an extended version of the following paper:
S. Lee and S. Kang, "Verifying a software architecture reconstruction framework with a case study," The Eighth International Conference on Software Engineering and Knowledge Engineering, San Francisco, USA, 2006.

architecture reconstruction has become the main stream of the reverse engineering research area (Bowman et al., 1999).

However, software architecture reconstruction is not as simple as traditional reverse engineering of source code; earlier researchers who have studied software architecture reconstruction with many case studies argue that human interventions are necessary during recovering software architecture (Rosso et al., 2005; Riva et al., 2002; Murphy et al., 2003). Software architecture reconstruction is not only to generate simple architecture view diagrams, but also to discover the design decisions of architect, and to provide detailed information for modifying, updating and redeveloping the software. In order to reconstruct the software architecture, the developers should know the technology and the domain model that have been applied to the software, as well as its quality attributes and functional requirements. With this knowledge, the architecture miner has to examine the architecture of an existing software system.

Accordingly, some researchers have suggested frameworks of software architecture reverse engineering (Clements et al, 2003; Deursen et al, 2004) and others have established toolset environments for the framework (Deursen et al., 2004; Gorton et al, 2005). Confronted with the challenge of recovering software architecture from just source code, it is important to understand how the developers can take all the necessary steps and succeed in achieving their goals by following in a step-by-step fashion, guided by the framework.

However, the existing frameworks are not mature enough to be able to help ordinary developers analyze software architecture for modifying or redeveloping from the existing software system in order to satisfy new requirements or eliminate deficiencies in it. Often previous frameworks were not clear about what is reconstructed. Some frameworks just focus on defining the steps of drawing structural diagrams without considering the purpose of reconstruction and appropriate viewpoints. For example, Symphony and QADSAR frameworks to be introduced in Section 2.1 do not demonstrate how to practice it utilizing software architectural views in software modification. On the other hand, when it comes to the reverse engineering tools to be introduced in Section 2.2, it is often the case that they do not ensure that the produced software architectural views are complete enough to assist modifying software.

In order to remedy this situation we propose a framework of software architecture reconstruction. This framework consists of a generic process for software architecture reconstruction and the steps to derive from it a concrete tool-based process to be used for actual architecture reconstruction. The architecture miner can use this framework to analyze source code for modifying source code as well as to reconstruct software architecture from source code.

The remainder of the paper is organized as follows: Section 2 reviews the previously known software architecture reconstruction frameworks, review reverse engineering tools and review software architecture reconstruction case studies. Section 3 develops our framework for reconstructing software architecture. In Section 4 we verify the framework by conducting a case study. Finally, Section 5 summarizes the contributions of this paper and describes the future research directions.

2. Related Work

In this section, we survey the previous research on software architecture reconstruction. Section 2.1 surveys the existing frameworks and the tools that support the frameworks. Section 2.2 explores reverse engineering tools. Finally, Section 2.3 examines two case studies, which offer lessons learned from the analysis of software systems.

2.1. *The Existing Frameworks for Software Architecture Reconstruction*

Several research groups have proposed frameworks for software architecture reconstruction. One group, Nokia, suggested Symphony, which is a reference framework to compare all activities related to software architecture reconstruction (Deursen et al., 2004; Deursen and Riva, 2004). Symphony outlines a view-driven software architecture reconstruction. Symphony consists of two phases: The first phase analyzes the problem for which architecture is needed and defines the required viewpoints and their mapping from source code. The second step extracts and analyzes information, applies mapping and creates views. Symphony provides a common reference framework to find and demarcate a research problem in software architecture reconstruction. However, it cannot be used directly by developers or researchers who desire to take actions step by step because it encompasses all possible steps, not providing practical steps to reconstruct software architecture for specific situations addressed by the paper (Stoermer et al., 2002).

Another group, SEI, suggested QADSAR, a framework that linked various system quality attributes to reconstructing an architectural view (Stoermer et al., 2003; Gorton et al., 2005). The QADSAR approach has five steps: scope identification, source model extraction, source model abstraction, element and property instantiation, and quality attribute evaluation. QARSAR is valuable because it is a systematic way to introduce a quality attribute driven perspective to software architecture reconstruction. However, QADSAR is limited in reconstruction capability because the framework does not utilize so called hypothetical views for accurate reconstruction of sophisticated architectures.

2.2. *Reverse Engineering Tools*

In this section we survey the existing reverse engineering tools. In Section 2.2.1, we first examine the tools that are associated with the two architecture reconstruction frameworks introduced in Section 2.1 and in Section 2.2.2 we classify the existing tools to static analysis tools and dynamic analysis tools.

2.2.1. *Tools associated with the existing frameworks*

Some researchers developed tools for software architecture reconstruction (Deursen et al., 2004; Gorton et al., 2005; Murphy et al., 2001; Kollmann et al., 2002). A research group in Nokia suggested an architecting environment (the ART environment) that facilitates architecture design, reconstruction, and maintenance in the entire life cycle of a software product line (Riva et al, 2004). Among the three toolsets constituting the ART environment, the second toolset for architecture is a reverse architecting toolset and contains three parts: the architecture knowledge base, the static view reconstruction tool and dynamic view reconstruction tool. Though the reverse architecting toolset is quite advanced, the toolset is confined to supporting a product line management. The tools in the ART reverse architecting toolset were developed independently from the Symphony framework.

The SEI group proposed a workbench approach that supports an extendible set of tools (Kazman 2003; Bass et al, 2003) named ARMIN (Architecture Reconstruction and MINing). ARMIN has four steps to draw architectural diagrams: (1) Information Extraction phase obtains information from various sources. (2) The Database Construction phase involves converting the extracted information into a tuple-based data format and loading it into a ARMIN tool. (3) The View Fusion phase combines information stored in the ARMIN database to generate a set of low-level views. (4) In the Architectural View Composition phase, the main work of building abstractions and representations and generating an architectural representation takes place. The ARMIN tool provides the ability to visualize and manipulate the set of views generated during reconstruction. ARMIN tool itself is not associated with QADSAR. Thus Gorton et al., 2005, who assessed the capabilities of software reverse engineering and architecture reconstruction tools based on QADSAR, utilized five reverse engineering tools including ARMIN and tried to employ appropriate tools for each step of QADSAR.

2.2.2. *Other Tools for Reverse Engineering*

Reverse engineering tools have evolved from extracting documentation from source code to constructing architectural information. As most current reverse engineering tools can be classified into the tools for static aspects and the tools for dynamic aspects, we will discuss about reverse engineering tools based on this dichotomy.

For static aspects, various kinds of reverse engineering tools analyze source code, extract source information, generate document and construct structural graphs such as flow charts, UML diagrams, and architectural views (Bellay et al., 1997; Sim et al., 2000; Murphy et al, 1998; Kollmann et al., 2002). Since static reverse engineering tools have been developed for long time, it is difficult to decide which one to select among them. Some useful tools are Understand for Java (Scientific Toolworks Inc.), Together (Boland Together), and ARMIN (Kazman et al, 2003, Gorton et al., 2005). Understand for Java

provides code metrics and code navigation with a variety of detailed graphical views. Together analyzes source code, generates UML class diagrams and sequence diagrams and synchronizes source code and UML diagrams. ARMIN constructs architectural views with scripting facilities.

For dynamic aspects, most development tools offer a tracking function and performance analysis tools show call graphs with performance-related information (Xie et al., 2002; Pacione et al., 2003). VTune (Intel Corp.), a famous performance analyzer, provides a call graph showing calling sequence with performance information. On the other hand, tools proposed in the area of dynamic reverse engineering are Shimba, AVID and DiscoTect. Shimba (Systä 2000) acquires trace information while programs are executing and automatically produces UML sequence diagrams and statechart diagrams. AVID condenses and visualizes software execution. DiscoTect (Yan et al., 2004) verifies that a system is consistent with its architectural design by observing the execution of the system.

2.3. *Case Studies*

There were several published reports of software architecture reconstruction case studies (Murphy et al., 1997, Bowman et al., 1999, Gröne et al., 2002, O'Brien et al., 2003, Stoermer et al, 2006). We found that among them three case studies are particularly relevant to our research on software architecture reconstruction framework.

The first one is the case study of Microsoft Excel by Murphy et al. Murphy et al's reconstruction has five steps: In the first step for high-level model definition, the focused aspects of the systems' structure are described. In the second step for source model extraction, a call graph or file dependency information is extracted. In the third step for defining map, the entities in the source and entities in the high-level models are related. In the fourth step of reflexion model computing, the user invokes a set of tools to construct a software reflexion model. In the fifth step for investigation and refinement, the reflexion model is compared against the source code to see if it is an adequate model as intended. The fourth and the fifth steps are iterated until the resulting reflexion model is found to be good enough.

The second one is the case study of Linux kernel by Bowman et al., which distinguishes the architecture in the developer's mind from the architecture built into a system and shows the differences between them. Bowman et al's approach is as follows: Firstly, they examined existing documentation. Secondly, they grouped source files into subsystems based on directory structure, naming conventions, and code comments. Thirdly, they extracted relations between the source files. Then they determined relations among subsystems. Finally, they used the relations to form software architecture of the system. Since these steps are useful for recovering design decisions when the architect is not

available, we incorporate these steps into our software architecture reconstruction framework to be presented in Section 3.

The third one is the case study of Apache by Gröne et al., which shows how students who have no domain knowledge learned Apache software architecture in an academic environment. Gröne et al. summarized the results of one-semester course of source code analysis on the structure of Apache 1.3. The students who had no domain knowledge about the system followed the steps below and presented the architecture successfully at the end of the course: The first step is to define the purpose of the analysis and explain key concepts of the system. The second step is to gather domain knowledge, understand the system and model the conceptual architecture of the system. The third step is to understand the function, the configuration and handling of the software product. The final step is to understand the implementation of the software product. We use these steps to establish our own framework for people having no domain knowledge of modifying open source.

3. FASAR: The Framework for Automating Software Architecture Reconstruction

In this section, we develop a software architecture reconstruction framework that was introduced in Section 1. We call our software architecture reconstruction framework FASAR, which stands for 'the Framework for Automating Software Architecture Reconstruction.' For this framework provides a generic software architecture process that is used to produce concrete tool-based processes and the concrete architecture reconstruction process that can have more and more automated steps as more tools become available for architecture reconstruction. The goal is to recover software architecture with as much help of tools as possible taking much shorter time than manual analysis would take.

The rest of Section 3 is organized as follows: In Section 3.1, we introduced the FASAR framework. In Section 3.2, we show the FASAR generic process. This is done by explaining how we derived the various steps of the FASAR generic process. In Section 3.3, we show how to obtain the tool-based architecture reconstruction process.

3.1. *The FASAR Framework*

The architecture reconstruction framework we propose in this paper consists of a generic process for architecture reconstruction and three main steps (Figure 1): The first step extracts system characteristics from the target system; the second step takes as input the generic architecture reconstruction process and the system characteristics, selects an appropriate set of tools for the various steps of the generics process thereby producing a specific toolset-based process for architecture reconstruction that fits the target system. In

third step the resulting process is actually applied to the target system to produce a static view of architecture and a dynamic view of architecture.

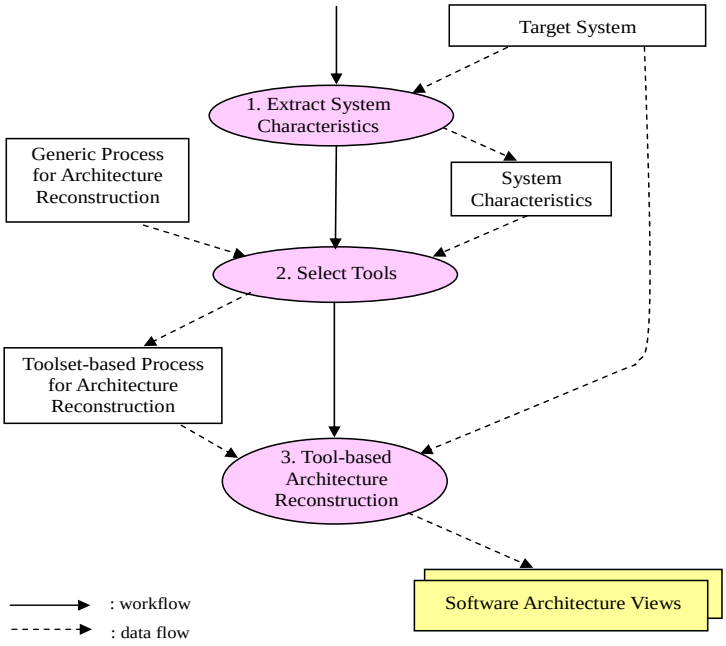

Fig. 1. Overview of the FASAR framework

3.2. *Generic Process for Software Architecture Reconstruction*

Some standard views such as code view or module view can be directly recovered from the source code, build files and/or configuration information. To recover other views, the architecture miner needs to execute the system. In either case, the architecture miner usually needs so called *hypothetical views*, e.g., hypothetical execution view or hypothetical conceptual view (Deursen et al., 2004). Hypothetical views are presumed architecture views against which the actual system is tested for architecture discovery. Outline of the generic architecture reconstruction process of the FASAR framework is depicted in Figure 2. In the remainder of Section 3.2 we elaborate and concretize the outlined process to derive the detailed generic process, which is developed by first identifying the necessary steps (Section 3.2.1) and then by connecting the steps with various artifacts that are required as input or produced as output (Section 3.2.2).

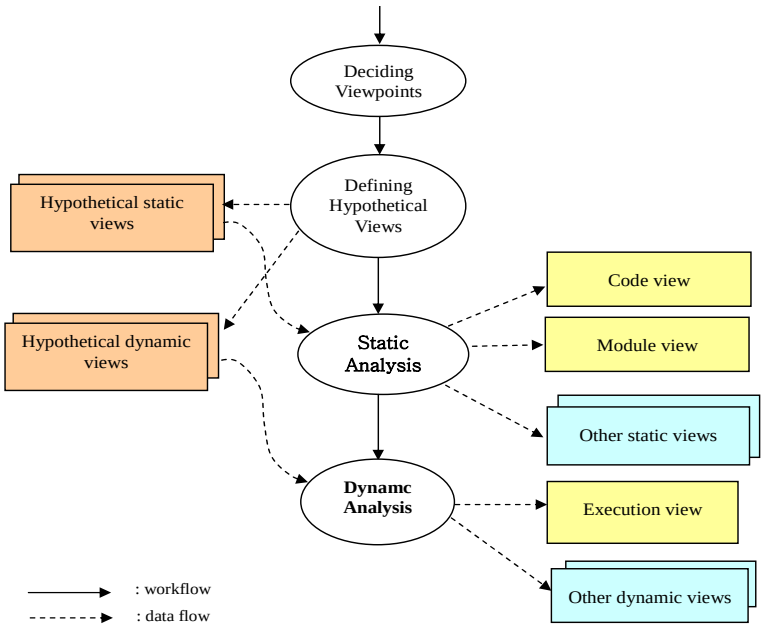

Fig. 2. Overview of the generic architecture reconstruction process of FASAR

3.2.1. *Deriving the Generic Process for Software Architecture Reconstruction*

In constructing the generic process of FASAR, we take the following approach: First, as its base we adopted Symphony because Symphony embraces both the preparation and the execution of the reconstruction. Reconstruction Preparation is required because in this phase developers attempt to understand problems in the existing software and the working mechanism of the software. Second, we base our reconstruction execution steps on QADSAR because QADSAR has the concrete steps for building architecture views. Thirdly, FASAR reflects the activities of case studies, such as Gathering Domain Knowledge, and Understanding the Configuration. These activities define more effective steps for analyzing a software system that do not require interviewing architects. The mapping results are shown in Tables 1 and 2. The existing frameworks do not explicitly separate static analysis and dynamic analysis. We think this distinction is fundamental since the necessary information and the techniques for them to apply are widely different although not disjoint. And therefore the distinction has been built into the framework explicitly.

Table 1. Deriving the Preparation Phase steps of FASAR generic process from the existing frameworks and case studies.

Symphony Framework	QADSAR Framework	Linux Case Study	Apache Case Study	RM Case Study	FASAR Framework
Reconstruction Design					Preparation Phase
Problem elicitation					Problem definition
			Define the purpose of analysis		Describe the purpose
	Scope identification				Identify the scope
Concept determination				High-level model definition	Hypothetical view definition
Identify useful viewpoints					Determine architecture reconstruction sources & target viewpoints.
Define target viewpoint					
Define source viewpoint[†]					
Define mapping rules[‡]					
			Gather domain knowledge and understand the system		Understand the system and gather domain knowledge
Determine role & viewpoints of hypothetical views		Form a conceptual architecture	Understand the function and handling of the system		Define hypothetical views

[†] I.e., determine what information will be needed in order to create target views. In the Symphony framework, "source" includes source code, build files, configuration information, documentation or traces.

[‡] Ideally it is a formal description of how to derive a target view from a source view. In FASAR this substep is performed during the reconstruction phase.

Table 2. Deriving the Reconstruction Phase steps of FASAR generic process from the existing frameworks and case studies.

Symphony Framework	QADSAR Framework	Linux Case Study	Apache Case Study	RM Case Study	FASAR Framework
Reconstruction Execution					Reconstruction Phase
					Static analysis
		Use the conceptual architecture to group source files into subsystems[§]		Defining map	Define mapping between the source model and the target views
Data Gathering (Collect data from a system's artifacts)	Source model extraction (for static aspects)	Extract relations between source files	Understand the implement ation of the software product (including behavior)	Source model extraction (from source code)	Extract a source model (from source code)
Knowledge Inference (Derive target view from source view)	Source model abstraction & visualization	Determine relations between subsystems		Reflexion model computing	Construct target views
		Form a concrete architecture			
Information Interpretation	Element & property instantiation			Investigation & refinement	Analyze the Architecture
	Quality attribute evaluation				
					Dynamic analysis
					Define mapping between the execution model and the target views
Data Gathering (Collect data from a system's artifacts)	Source model extraction (for dynamic aspects)		Understand the implement ation of the software product (including behavior)		Extract an execution model
Knowledge Inference (Derive target view from source view)	Source model abstraction & visualization				Construct target views
Information Interpretation	Element & property instantiation				Analyze the Architecture
	Quality attribute evaluation				

[§] This substep is very specific to the case study and thus is not included in FASAR steps.

3.2.2. *The Details of the Generic Software Reconstruction Process*

The steps of FASAR generic process are grouped into the Preparation Phase steps and the Reconstruction Phase steps. The Preparation phase analyzes problems for which architecture is needed and defines the conceptual architecture of the system (Deursen et al., 2004; Deursen and Riva, 2004; Bowman et al., 1999; Gröne et al, 2002). For that it is necessary to perform two tasks: Problem Definition and Conceptual Architecture Determination. The Reconstruction Phase extracts and analyzes information, defines and applies mapping between source and conceptual model, and creates two kinds of views: static views and dynamic views. The outputs from the Preparation Phase become the inputs to the Reconstruction Phase. A scenario, describing current and expected situations in the Preparation Phase, should be an input to the Reconstruction Phase, in order to determine which execution trace should be recorded in the Dynamic Reconstruction View task. Conceptual Architecture, defined in the Preparation Phase should also be an input to the Reconstruction Phase, in order to customize a module view of an abstract source model in the Static View Reconstruction task. In Reconstruction Phase, Static View Reconstruction analyzes source code and generates a Code View and a Module View. The final result of Static View Reconstruction becomes an input to the Dynamic View Reconstruction in order to aggregate detailed modules to abstract modules of Concrete Architecture. Finally, the Reconstruction Phase produces the three views of the software architecture of the target system: Code View, Module View and Execution View (Clements et al., 2003; Lattanze 2005). The details of the generic architecture reconstruction process of the FASAR Framework are depicted in Figure 3.

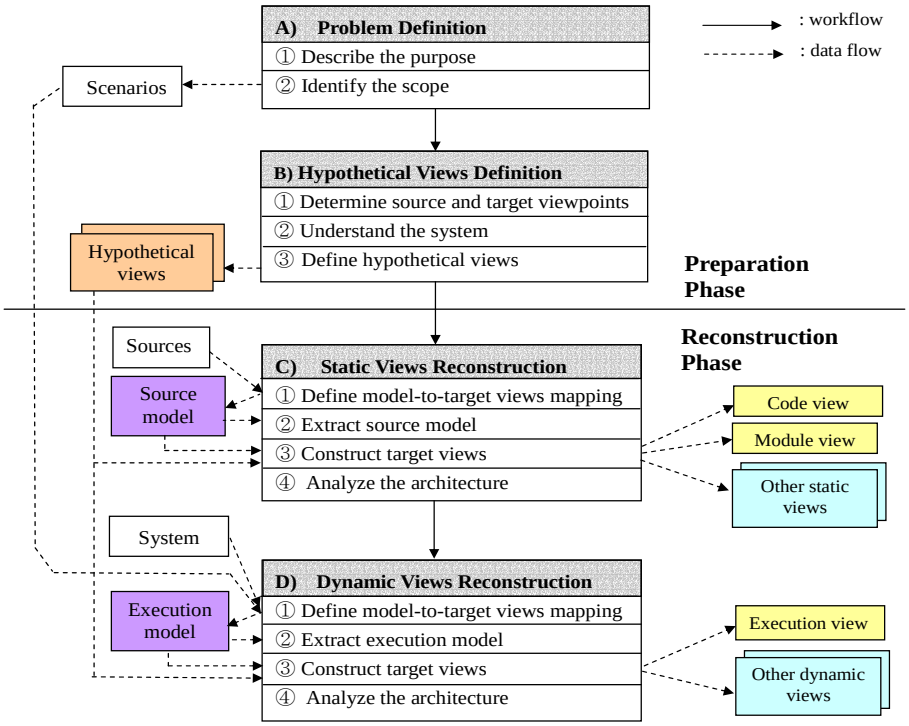

Fig. 3. The FASAR Generic Process for Software Architecture Reconstruction

A) Problem Definition

The architecture miner should select the essential part, identify mapping rules, identify the property to be discovered in the software and mine the specific part of the software until there is enough information for a modification during static analysis. The architecture miner should also select key scenarios, identify mapping rules, identify the property to be discovered and mine the specific execution of the software. In the Problem Definition step, we analyze the problem triggering the reconstruction with stakeholders. It consists of the following two substeps:

① **Describe the purpose:** The architecture miner studies the system to find compelling reasons to start reconstruction.

② **Identify the scope:** The architecture miner identifies the required viewpoints, the parts of the system to be reconstructed and the relevant quality attribute scenarios.

B) Hypothetical Views Definition

While the architecture miner should use the software to understand the problem for the software, the architecture miner he should also define the current architecture of the system by studying the domain model of the software and technologies applied to the

software. Conceptual hypothetical architecture defined in the step is later used in abstracting source model and execution model in the Reconstruction Phase. Gathering domain knowledge and understanding the handling of the software would allow the architecture miner to discover the properties implemented in the software. Hypothetical Views Definition step consists of the following three substeps:

① **Determine source and target views:** The architecture miner decides what kind of views to obtain from software artifacts such as source code and software execution.

② **Understand the system:** The architecture miner finds out how to configure and administer the software and how to utilize the API.

③ **Define hypothetical views:** The architecture miner draws the hypothetical architecture by understanding the mechanisms of the relevant parts.

C) Static Views Reconstruction

The architecture miner should analyze code and build the concrete architecture in a module view (Clements et al., 2003; Bowman et al, 1999; Gröne et al., 2002; Murphy et al., 1997; Murphy 2003). It consists of the following four substeps:

① **Define model-to-target views mapping:** The architecture miner defines the tentative relationships between the source model and hypothetical views.

② **Extract source model:** The architecture miner extracts source elements from available source code files.

③ **Construct target views:** Source level information is abstracted to the abstraction level of the target view. The architecture miner should identify and apply abstraction strategies.

④ **Analyze the architecture:** The target views constructed may not be in the form of selected viewpoints or in the form directly suitable for addressing the problem at hand. Then they have to be interpreted or presented in the form that is comprehensible to the stakeholders. Additionally, the architectural evaluation may be performed on the resulting architecture (Kazman 2003).

D) Dynamic Views Reconstruction

The architecture miner should analyze execution traces of the software and construct an abstract execution model such as Component and Connector view (Clements et al., 2003; Rosso 2004; Riva et al., 2002; Murphy et al., 1997). It consists of the following four substeps:

① **Define model-to-target views mapping:** The architecture miner define the tentative relationships between the source model and hypothetical views.

② **Extract execution model:** The architecture miner extracts the real interaction of software elements from software execution.

③ **Construct target views:** The architecture miner abstracts details of real interaction of software elements.

④ **Analyze architecture:** This substep is the same as C) ④**.**

3.3. *The Three Steps of the FASAR Framework*

In Section 3.2, we showed the generic process for architecture reconstruction. Together with the target system, this is the two initial inputs to the three main activities of the FASAR framework as shown in Figure 1. In this subsection, we explain in detail the three steps of the FASAR framework one by one.

3.3.1. *Extracting the System Characteristics*

To determine the necessary tools, it is crucial to know various system characteristics since many tools are system dependent. Particularly important system characteristics for software architecture reconstruction include: the programming language, development environment, execution environment and the feature to be investigated. In this step, the architecture miner identifies the criteria to select candidate tools supporting the system characteristics.

3.3.2. *Deriving the Toolset-based Process for Architecture Reconstruction*

In the second major step of the FASAR framework, the architecture miner derives a concrete tool-based process. In order to do that the architecture miner should know exactly what steps and/or substeps of the generic process those tools can be applied to. In this section, we illustrate this step by working with a set of tools for static steps and dynamic steps. For that we consider twelve widely used tools as shown in Table 3. The tools in Table 3 are all based on the Java language and when tools should support a different language we would have complete different set of candidates tools.

Then we map the functions of the tools to the steps of Figure 3. The mapping is needed to determine which steps are supported by which tools. For each step defined in Figure 3, we check whether it can be automated. If the step does not require human intervention, the step is automatable (marked with ‘O’ in Table 4). If a step requires human intervention but the human intervention can be supplanted by a default setting, the step may be automated but the automation could not produce the expected result (marked with ‘Δ’ in Table 4).

The mapping table lets us know exactly which tools can be used for which steps when we conduct architecture reconstruction. More than one tools per step may need to be selected because, even when they support the same step, the actual functions and results of the tools may be different. In addition, mapping of a tool to a step does not mean that the step is fully automated. It means only that the step is supported by the tool. After reviewing the steps of the Preparation Phase, we conclude that Preparation Phase largely relies on human intelligence. On the other hand, the major portion of the Reconstruction Phase can be automated but the phase still requires human intervention between steps. The results are presented in Tables 4 and 5. Although we selected twelve tools and used them as an illustration, all of these tools are widely available and the comparisons in Tables 3, 4 and

5 can be conveniently used for the architecture miner as the starting point or a model for their tool selection and tool usage analysis as is used in the case study of Section 4.

Table 3. A sample list of software architecture tools that support Java

Tools for Static Architecture Reconstruction	Doxygen	A documentation tool for several languages. Analyzes the source code and generates on-line documentation browser in HTML or a reference manual in MS-word, PDF, etc.
	Together Developer	A tool to analyze source code and generates UML class diagrams and sequence diagrams. Also synchronizes the source code and UML diagrams, thus enabling developers to maintain the diagram consistent with the code.
	Reflexion Model(RM)	Analyzes the source code of a software system from the viewpoint of a particular high-level model that developers define with Doxygen. The approach provides a solution to the problem that high-level models are almost always inaccurate with respect to the system's source code.
	Understand for Java	A tool for reverse engineering, documentation, code exploration and metrics tool for Java source code. Analyzes source code and provides a variety of graphical reverse engineering view, code navigation using a detailed cross reference and many different code metrics.
	PBS	Provides software structure based on the hierarchic decomposition of the software system into subsystems. Consists of three tools: cfx extracts function call and variable access relations from source code. grok is used to determine relations between source files based on the result of cfx. lsedit visualizes the extracted system structure.
	ARMIN	Extracts source information from the source code, customizes it by using a set of command scripts, and builds abstractions of the information to obtain various views of the architecture.
Tools for Dynamic Architecture Reconstruction	HPROF	A simple command line profiling tool for generating trace data and profiling heap and CPU utilization. Included in JVM native library, and used to capture data by other graphic tools.
	VTune	A performance analyzer to provide a call graph, showing calling sequence and graphically displays the critical path, and various performance information. Supports almost all applications on all sizes of systems based on Intel® processors.
	Jinsight	A free profiling tool provided by IBM. Generates Message Sequence Chart (MSC) from profiling data.
	Shimba	Automatically produces Sequence Diagrams for Java programs. Trace information is acquired while programs are executed, which then is used to create statechart diagrams and sequence diagrams.
	AVID	Developed to visualize software execution in the view of software architecture. Condenses the system's execution in terms of user-defined components.
	DiscoTect	A tool to construct a software architecture view from monitoring software execution. Specially advanced in mapping detailed execution to architectural events.

Table 4. Steps to apply static tools

Steps	Doxygen	UFJ	Together	PBS	RM	ARMIN
Source Model Extraction						
Make a table of all files	Δ	Δ	Δ	O	O	O
Select the essential part		Δ	Δ	Δ	Δ	Δ
Extract concrete relationships	Δ	Δ	Δ	O	O	O
Source Model Abstraction						
Identify mapping rules				O	O	O
Identify concrete relationships				Δ	Δ	Δ
Establish concrete architecture						Δ
Implementation Analysis						
Identify Property						
Examine the architecture						Δ

Table 5. Steps to apply dynamic tools

Steps	HPROF	VTune	Jinsight	Shimba	AVID	DiscoTect
Execution Model Extraction						
Identify key scenarios						
Extract trace data	Δ	Δ	O	O	Δ	O
Draw execution flow		Δ	Δ	Δ	Δ	
Execution Model Abstraction						
Identify mapping rules					O	O
Abstract execution flow						
Conduct measurements		Δ				
Establish concrete architecture					Δ	Δ
Execution Analysis						
Identify Property						
Examine concrete architecture						

3.3.3. *Conducting Toolset-based Architecture Reconstruction*

Once a toolset-based architecture reconstruction process has been established as in Section 3.3.2, finally the architecture miner carries out the steps in Figure 3 as explained in detail in Section 3.2.2 with the help of tools as determined in Tables 4 and 5. Note that depending on whether the architecture miner is interested in static views or dynamic view or both, respectively, only one of Tables 4 or 5 or both need be used. It is important to note, however, that when an applicable tool has been identified for a step, still the outcome of applying the tool may not give the results exactly in the desired format. In such a case, manual conversion is necessary.

4. A Case Study: Application of the FASAR Framework to the Reflexion Model Software

In this section, we perform a case study of applying the FASAR framework. The main purpose of the case study is to demonstrate that the framework can be used to reconstruct software architecture for systems with industry level size and complexity. In addition, the case study helps developers understand our framework and provides a guideline for developers to follow to apply the FASAR framework. We selected as the target system the source code for one of the architecture reconstruction tools, the Reflexion Model (Murphy et al., 1997; Murphy 2003).

4.1. *Extracting the System Characteristics*

The software was implemented in Java 1.3 and Eclipse 2.1.1 on MS Windows. It consisted of 76 files and the total number of lines was 13,996 and the ratio of comments to code was 0.34.

4.2. *Selecting Tools*

Since the tools shown in Table 3 are all Java-based, they make a good list of candidate tools. However, for static analysis, we selected only Doxygen, Together, and Reflexion Model based on the availability. Other tools were not selected either because the tool does not provide a convenient graphic user interface or because the selected tools already provide more appropriate information. Doxygen is used in the step of making a table of all files because the tool produces the file list of packages. Together is used in the step of selecting the essential part because the tool generates the class diagrams of packages so that we can refer to the class diagrams to discover important parts. Reflexion Model is used in the step of identifying mapping rules and establishing concrete architecture because the tool supports a user in identifying the mapping relationships between source code and an abstract model and generates the relations of the abstract model. For dynamic analysis, we selected only HPROF. Almost all advanced tools such as Jinsight (Pacione et al., 2003), Shimba (Systä 2000) and AVID (Pacione et al., 2003; AVID) were unavailable to us. Another reason was that other dynamic analysis tools are very sensitive to their environment; those tools did not support the eclipse platform and so cannot be used to trace the execution of the target software system. HPROF is used in tracing the target system because it can be applied to any software systems implemented in Java and running on JVM.

4.3. *Toolset-based Architecture Reconstruction*

4.3.1. *Preparation Phase*

A) Problem definition

① **Describe the purpose:** In this step, we collected and read technical papers about Reflexion Model (Murphy et al., 1997; Murphy et al., 2001; Murphy 2003) to learn how to operate the system. Then we prepared the environment and installed the software. Next we used the software to analyze a compiler that translates TTCN code to C Code. Finally, in the experiment we discovered better ways to use the software Doxygen.
② **Identify the scope:** In this step, we prioritized the five improvement items and selected the first one: supporting hierarchical high-level modeling. Then we specified the following use-case scenarios to support hierarchical high-level modeling:
a) When defining a high-level model, the user chooses a rectangle icon from the menu of the high level model. Then, the user draws a rectangle representing the upper modules embracing lower modules shaped in ellipses
b) When defining mapping, the user adds the mapping relationships between the lower modules and the upper modules to the map.
c) When computing mapping, the tool computes hierarchical mappings between several layers, and then presents the hierarchical model.

B) Hypothetical Views Definition
① **Determine source and target views:** In this step, we identified aspects of the target system. When we specified use-case scenarios to identify the scope, we identified a view showing the execution flow among important modules. Thus we identified a module as a collection of classes and a line as a call among modules.
② **Understand the system:** In this step, we learned how to configure and administer the product and how to utilize the API. We had to study and experiment with the working mechanism and the graphic packages for Eclipse plug-in software before we started configuring. After that, we imported the source code as an Eclipse plug-in project into the Eclipse tool, and created the executable file of the Reflexion Model (RM). For studying the API, we analyzed the main file and grasped the four classes connected to the user interface of the RM program. We made the four classes as starting points of the analysis of the RM source code
③ **Define hypothetical views:** Figure 4 depicts the hypothetical architecture that represents the main functions of the Reflexion Program. The Extraction part parses the source code and extracts detailed source information. The Modeling part supports the user drawing the high level model, then converts the diagram into a text file. The Mapping part parses the text file that holds the information of the high level model and mapping rules. The Calculation part maps the relationships among source elements to the relationships among modules of high level model. The Presentation part shows the result of the calculation in a graphic form.

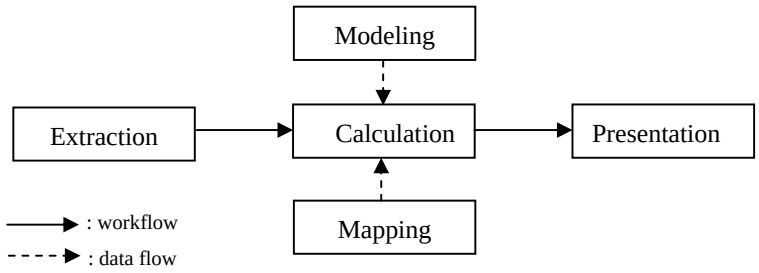

Fig. 4. Hypothetical view of Reflexion Model

4.3.2. *Reconstruction Phase*

The Reconstruction Phase extracts and analyzes information, defines and applies mapping between the source and the conceptual model, and creates concrete (instances of) architecture views. We began with the directory structure, and modified mapping rules several times until we had captured the proper mapping between the hypothetical architecture and source elements. Finally, we were able to create concrete architecture (Figure 5). A number on an arc between two modules represents the number of call relationships between those two modules.

C) Static Views Reconstruction

① **Define model-to-target views mapping:** In this step, we identified the mapping rules between source code and a source model. The initial modules of a source model are the same modules of the hypothetical view and we began the mapping the directory structure of the source code to the initial modules of a source model. The mapping rules were modified as a source model are extracted and reviewed in the step of extracting a source model.

② **Extract source model:** In this step, we executed Doxygen that showed a directory structure, a list of files, classes, etc. Then we edited the result to obtain Table 5, which includes the list of files. Next, we executed Together to grasp class diagrams. However, the tools did not extract call relationships among source elements, except for inheritance relationships. We noticed that some tools extract and abstract the relationships simultaneously. Thus we skipped the step for extracting the concrete relationships to the next step of Source Model Abstraction and expected that abstraction tools would help.

Table 5. Code Structure of Reflexion Model

/Doxygen/RM/jrmtool/sec/ca/ubc/cs				
jRMTool	compute	*.java (9)		
	eclipse	graph	spline	*.java (3)
			*.java (10)	
		gui	plugin	*.java(15)
			*.java (1)	
	map	*.java (1)		
	model	*.java (3)		
	struct	*.java (5)		
	util	*.java (10)		
sugiyama	algorithm	*.java (12)		
	model	*.java (4)		
	*.java (4)			

③ **Construct target views:** In this step, we utilized RM for showing the target views that exhibits actual relationships among modules. A difference between the hypothetical architecture (Figure 4) and the target view (Figure 5) is that there are no definite call relationships between the Calculation module and the Extraction, Modeling, and Mapping modules in the concrete architecture; The symbols, ❶❷❸, denote no call relationships exist among the modules in Figure 4. Since our target source has graphic user interface and event handling mechanisms, the relationships between events are implicit.

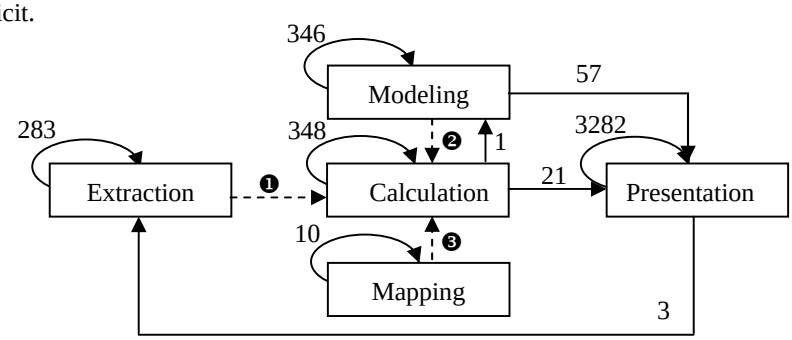

Fig. 5. A static view of Reflexion Model

④ **Architecture Analysis:** In this step, as the purpose for architecture reconstruction was to enable the Reflexion Model to support a hierarchical high-level modeling, we located the parts of source code to be modified. Then we examined the source code and selected important classes and functions related to the modification of a hierarchical high-level modeling.

D) Dynamic View Reconstruction

① **Define model-to-target views mapping:** In this step, we introduced the same mapping rules that we used in abstracting a source model.

② **Extract execution model:** In this step, we used the scenario c) described in Problem Definition task to extract a trace file from the execution of the function, "compute Reflexion Model," by using HPROF. As a result, we created a trace file with 150,796 lines. The trace data was not accurate in the call relationships between functions because HPROF just recorded execution order and did not mark when a function starts and finishes.

③ **Construct target views**

Following the mapping rules, we abstracted the trace data of the "Execution Model Extraction" to the upper level, as in Figure 5. A number on an arc between two modules represents the number of execution orders between two modules. Then we discovered that the modules that had showed no relations in the static analysis showed some relations in the dynamic analysis. We analyzed an execution order of Reflexion Model by importing and manipulating the trace data in MS Excel. As a result, we discovered that the implicit execution order of modules in a Static View becomes explicit, and we were able to see the flow from Extraction to Calculation.

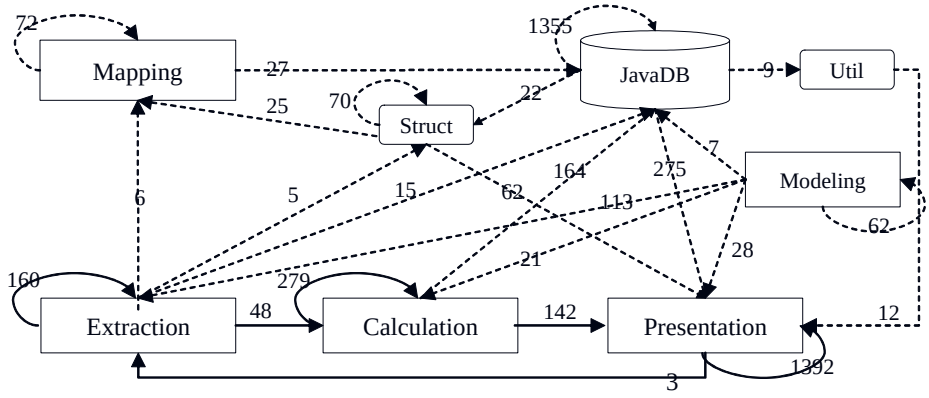

Fig. 6. A dynamic view of Reflexion Model

④ **Analyze the architecture:** In this step, we first identified what we need to know in the source code, referring to the concrete scenario from the Problem Definition task, "Computing Reflexion Model." Then we checked which parts are affected by the scenarios when we modify the calculating algorithm to support hierarchical mappings among several layers. Finally, we examined the source code and found the important parts related to the modification requirements.

5. Conclusion

In this paper, we proposed the FASAR framework for reconstructing software architecture. We utilized Symphony as the base framework, and then incorporated

QADSAR into it. The Symphony framework includes the steps for reconstruction preparation such as purpose definition, conceptual architecture, etc., which are important steps for achieving reconstruction engineering process that fits the purpose. By incorporating such steps, we made FASAR produce results that are consistent with the goals. We also took concrete steps from QADSAR. Moreover, we have integrated the procedures used in some revealing case studies.

FASAR is a practical framework in that (1) the framework is presented in terms of a generic process so that binding the process and tools can be made depending on their availability of tools and the characteristics of the system; (2) by classifying architecture views into static views and dynamic views and building it into the FASAR framework, our FASAR framework guides its users in approaching architecture reconstruction in a way suitable to each category of architecture views; and (3) the architecture reconstruction process is presented in a step-by-step and easy to follow manner.

We verified our FASAR framework through the case study of Reflexion Model. During the case study, we found and repaired several defects in the early versions of FASAR. In addition, the case study provides a practical guideline that helps developers understand and utilize our framework by showing a concrete example of a software system. Developers can find examples of a conceptual model, a concrete model and several views embodied in the case study.

The following are future research directions. First, we should survey tools more thoroughly including recent advanced tools and experiment with them. There are many reverse engineering tools but the tools we used were limited to the tools for the Java language. Many research papers report the result of reverse engineering in Java but many of the tools discussed in those research papers have not been published yet. Second, we should perform more case studies in order to consolidate our framework. Our case study of Reflexion Model targeted a specific environment and a specific subject. Performing more case studies for software systems from various domains would confirm that the FASAR framework is indeed widely applicable. In the long term, we want to develop our own toolset to reconstruct software architecture for modifying source code that is based on the FASAR framework.

References

1. AVID, *AVID*, http://www.cs.ubc.ca/~murphy/AVID/.
2. L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice, 2nd Ed.*, Addison-Wesley, 2003.
3. B. Bellay and H. Gall, “A comparison of four reverse engineering tools,” Proc. 4th Working Conference on Reverse Engineering, pp. 2-11, The Netherlands, 1997.
4. Boland, *Together*, http://www.borland.com/together/ developer/.
5. T. R. Bowman, C. Holt and N. V. Brewster, “Linux as a case study: its extracted software architecture,” Int’l Conf. on Software Engineering, Los Angeles, 1999.

6. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2003.
7. A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen and C. Riva, "Symphony: view-driven software architecture reconstruction," Proc. 4th Working IEEE/IFIP Conf. on Software Architecture, 2004.
8. A. van Deursen and C. Riva, "Software architecture reconstruction," Proc. 26th Int'l Conf. Software Engineering (ICSE), p. 745 – 746, May 23-28, 2004
9. *Doxygen*, http://www.doxygen.org/.
10. I. Gorton and L. Zhu, "Tool support for Just-In-Time architecture reconstruction and evaluation: an experience report," Proc. 27th Int'l Conf. on Software Engineering, 2005.
11. B. Gröne, A. Knöpfel and R. Kugel, "Architecture recovery of Apache 1.3 - A case study," Proc. Int'l Conf. on Software Engineering Research and Practice, pp. 87-93, Las Vegas, 2002.
12. Intel Corp., *VTune*, http://www.intel.com/cd/software /products/asmo-na/eng/vtune/index.htm.
13. R. Kazman, L O'Brien and C. Verhoef, "Architecture Reconstruction Guidelines, 3rd Ed.," CMU/SEI-2002-TR-034, 2003.
14. R. Kollmann, P. Selonen, E. Stroulia, T. Systä and A.Zündorf, "A study on the current state of the art in tool-supported UML-based static reverse engineering," Proc. 9th Working Conference on Reverse Engineering, 2002.
15. A.T. Lattanze, "The Architecture Centric Development Method," CMU-ISRI-05-103, 2005.
16. G. C. Murphy and D. Notkin, "Reengineering with Reflexion Models: A Case Study," *IEEE Computer 30, 8*, pp.29-36, 1997.
17. G. C. Murphy, D. Notkin and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap Between Design and Implementation," *IEEE Transactions on Software Engineering*, 27(4):364--380, April 2001.
18. G. C. Murphy, *Reflexion Models*, http://www.cs.ubc.ca/ ~murphy/jRMTool/doc/, 2003.
19. L. O'Brien and V. Tamarree, "Architecture Reconstruction of J2EE Applications: Generating Views from the Module Viewtype," CMU/SEI-2003-TN-028, 2003.
20. L. O'Brien and C. Stoermer, "Architecture Reconstruction Case Study," CMU/SEI-2003-TN-008, 2003.
21. M. J. Pacione, M. Roper and M. Wood, "A comparative evaluation of dynamic visualisation tools," Proc. 10th Conference on Reverse Engineering, pp. 80-89, 2003.
22. C. Riva and J. V. Rodriguez, "Combining static and dynamic views for architecture reconstruction," Proc. 6th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, pp. 11-13, Budapest, Hungary, 2002.
23. C. Riva, P. Selonen, T. Systa, A.-P.Tuovinen, Jianli Xu and Yaojin Yang, "Establishing a software architecting environment," Proc. 4th Working IEEE/IFIP Conference on Software Architecture, pp. 188-197, June 2004.
24. C. D. Rosso, "Performance analysis framework for large software-intensive systems with a message passing paradigm," Proc. 2005 ACM symposium on Applied Computing, Santa Fe, New Mexico, 2005.
25. Scientific Toolworks Inc., *Understand for Java*, http://www.scitools.com/uj.html.
26. S. E. Sim and M.-A. D. Storey, "A structured demonstration of program comprehension tools," Proc. 7th Working Conference on Reverse Engineering, pp. 184-193, 2000.
27. C. Stoermer, L. O'Brien and C. Verhoef, "Practice Patterns for Architecture Reconstruction," Proc. 9th Working Conference on Reverse Engineering, pp.151-160, Virginia, USA, 2002.
28. C. Stoermer, L. O'Brien and C. Verhoef, "Moving towards quality attribute driven software architecture reconstruction," Proc. 10th Working Conference on Reverse Engineering, pp. 46-56, 2003.
29. C. Stoermer, A. Rowe, L. O'Brien and C. Verhoef, "Model-centric software architecture reconstruction," *Software—Practice & Experience, Vol. 36, Issue 4*, April 2006.

30. Sun, *HPROF*, http://java.sun.com/developer/technical Articles/Programming/HPROF.html.
31. T. Systä, "Understanding the Behavior of Java Programs," Proc. of the 7th Working Conference on Reverse Engineering, Brisbane, Australia, 2000.
32. T. Xie and D. Notkin, "An Empirical Study of Java Dynamic Call Graph Extractors," Technical Report UW-CSE-02-12-03, (Seattle, WA), December 2002.
33. H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich and R. Kazman, "DiscoTect: A System for Discovering Architectures from Running Systems," ICSE 2004: 470-479, 2004.