Reverse Engineering with Reflexion Model: TTCN Translator

Team TTA / Rolling Final Project

May 5, 2005

Team TTA / Rolling:

ChangSup Keum

ChangSun Song

JungHo Kim

SeonAh Lee

ShinYoung Ahn

Table of Contents

1. Introduction ... 3

2. Choosing the Right Tool ... 3

3. Reflexion Model: Tool and Technique ... 4

4. Reflexion Model: Experimental Setup ... 5

5. Reengineering with Reflexion Model ... 6

- 5.1 The result: the first trial ... 6
- 5.2 The result: the second trial ... 11
- 5.3 The third refinement ... 13

6. Evaluation on Reflexion Model ... 17

- 6.1 Trial and errors ... 17
- 6.2 Evaluation statistics ... 18

7. Additional Experiment ... 19

- 7.1 Experiment Setting ... 19
- 7.2 Experimental Result ... 23
- 7.3 Experimental Summary ... 25

8. Possible Improvements ... 25

- 8.1 Supporting hierarchical High Level Modeling ... 25
- 8.2 Looking into the target code ... 25
- 8.3 Auto-generation of first High Level Model ... 26
- 8.4 Using meta model to describe the architectural styles ... 26

9. Conclusion ... 26

Reference ... 27

1. Introduction

In our MSE studio project, we have been developing TTCN (Tree and Tabular Combined Notation) Translator system which translates from TTCN-MP (Machine Process-able) to ATS (Abstract Test Suite) written in ANSI C code. TTCN is a language which has been used to specify test cases for many kinds of applications, including mobile communications, wireless LANs, cordless phones, Broadband technologies, CORBA-based platforms, and Internet protocols. We have developed the parser and tree of TTCN Translator by using JavaCC/JJTree tools because these tools generate the parser and tree code semi-automatically.

Even though we have got about 33,000 lines of code generated from JavaCC/JJTree, we don't know the internal structure of auto-generated code. Therefore, we should reengineer the auto-generated code. To analyze the code, we chose a tool: Reflexion Models, Reflexion Models allow us to begin with a structural high-level model that we can selectively refine to rapidly gain task-specific knowledge about the source code. Moreover, this tool would help us to refine an architectural view of the system and investigate the connection between the architectural component and source code. Moreover, it would increase our understanding of the code generated from JavaCC and lessen the danger of our reasoning in term of the high-level model alone.

In this paper, we will explain how Reflexion Model is used to analyze source code, explain the tool setup procedures and show you our reverse engineering activities. Finally, we will evaluate Reflexion Model tool and suggest what can be improved in the Reflexion Model.

2. Choosing the Right Tool

When we decided to use a reverse engineering tool to verify that the source code is in compliance with architecture, we found three reverse engineering tools in the tool list: Rigi, Lackwit, and Reflexion Models. Rigi is a tool for understanding large information spaces such as software programs, documentation, and the World Wide Web. It models the system by extracting artifacts from the information space, organizing them into higher level abstractions, and presenting the model graphically. Even if Rigi has a lot of useful features, we can't choose it because it is not able to support the system written in Java. Lackwit is a tool that helps programmers with reverse engineering or restructuring tasks. However, this tool focuses on detecting abstraction violations, identifying unused variables, functions, and fields of data structures, and detecting simple errors of operations on abstract data types (such as failure to close after open). Reflexion Models is selected as the reverse engineering tool because it is the most recent reverse engineering tool

which supports Java and it focuses on getting a high level architecture model.

3. Reflexion Model: Tool and Technique

Reflexion Model analyze the source code of a software system from the view-point of a particular high-level model [2]. The approach is a solution to the problem that high-level models are almost always inaccurate with respect to the system's source code. In this approach, an engineer defines a high-level model and specifies how the model maps to the source. Then, the Reflexion Model Tool computes Reflexion Model that shows where the engineer's high-level model agrees with and where it differs from a model of the source [2]. Figure 1 illustrates the overall approach of the Reflexion Model.

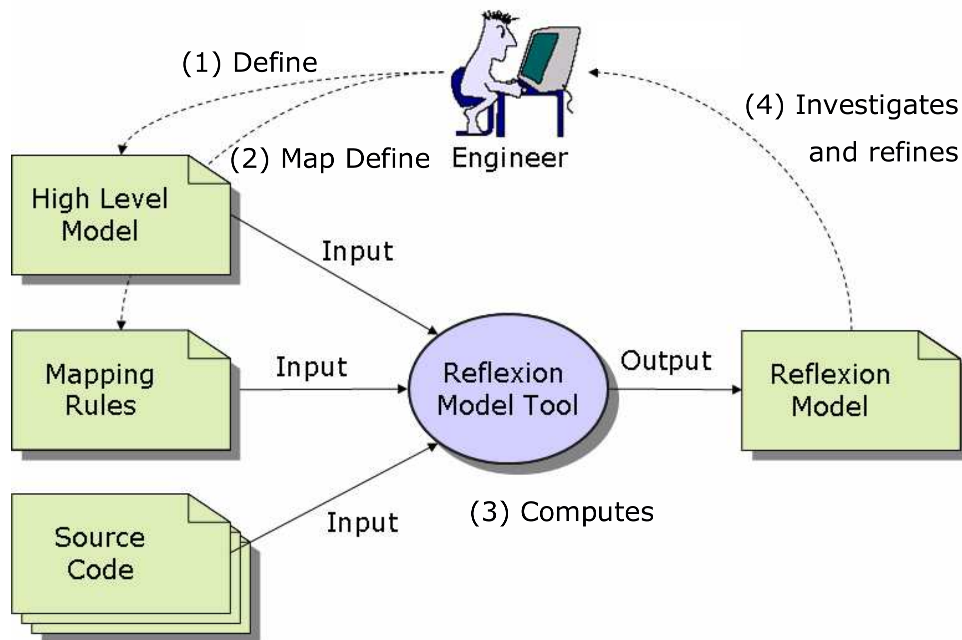

Figure 1. The Reflexion Model Approach

The important point of Reflexion Model approach is that it is lightweight and iterative. The user can easily and rapidly access the structural information of interest and can balance the cost of refinement with the potential benefits of a more complete and accurate model.

To derive a Reflexion Model from source code and iteratively refine it, the user performs four steps, as Figure 1 shows. These steps are repeated until the user gets the detailed model.

(1) Define high-level model

The high-level model describes aspects of the system's structure that helps reason the refinement task at hand. This step may involve reviewing artifacts(source code, document), interviewing experts,

or looking at similar architectures

(2) Define map

The user defines a map that describes which entities in the source code and high-level models relate.

(3) Compute Reflexion Model

The tool computes the Reflexion Model from the defined high-level model, a defined map, and the source code. The Reflexion Model lets the user see interactions in the source code from the view point of the High Level Model.

(4) Investigate and refine

The user can investigate the system by viewing a displayed Reflexion model. But the displayed Reflexion Model is not sufficient to provide detailed information. The user should analyze the Reflexion Model information which are mapped to particular arcs in the Reflexion Model and unmapped values. After analyzing these information, the user refine the high-level model.

4. Reflexion Model: Experimental Setup

This section introduces the most successful setup environment in which the Reflexion Model analysis tool can operate its full functionalities. The Reflexion Model analysis tool is highly dependent upon the versions of Java Runtime Environment and the Eclipse platform.

- Java Runtime Environment: J2SDK 1.4.0.5
- Eclipse Platform: Eclipse 3.0.2
    - Required Plug-ins
        - org.eclipse.ui
        - org.eclipse.draw2d
        - org.eclipse.core.resources
        - org.eclipse.jdt.core

After setting up this environment, you can use the Eclipse update manager to install the Reflexion Model Tool plugin. The update site for the plugin is “http://www.cs.ubs.ca/~murphy/jRMTool/eclipse/updates” [3]. After installing the plug-in software, you can refer to “Reflexion Model Tool Guide” by clicking Help > Help Contents on the menu of the Eclipse.

In addition, the Reflexion Model analysis tool requires the resource structure. The required resources include a rmt file and the target source code that the tool is going to analyze. A rmt file should be located in a Java project that can be made by the wizard of the Eclipse IDE, and the target source code should be located in a folder that is located under the project location. For example, if *test.rmt* file is located

at *../Test/test.rmt*, bundle of source code should be located under *../Test/src/.*

5. Reengineering with Reflexion Model

5.1 The result: the first trial

5.1.1 High Level model

TestGen system is a kind of a compiler which translates TTCN code to ANSI-C code. We already knew the concept of a compiler and the fact that JavaCC merges the lexer into parser. Thus, we added five nodes: Parser, AST(Abstract Syntax Tree), SymTab(Symbol Table), CodeGen(Code Generator), and Semantic. Next, we added arcs between nodes as Figure 2.

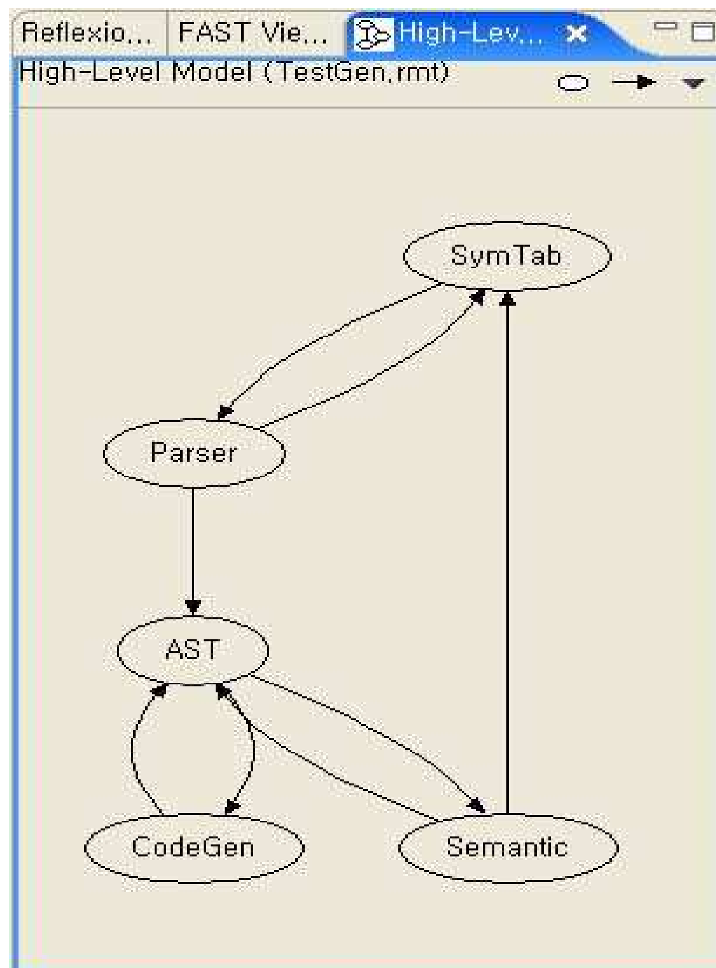

Figure 2. High Level Model

5.1.2 The Classes of Source Code

The TTCN parser generated by JavaCC amounts to 40,000 lines of code and consists of the following classes in Table 1.

Table 1. Classes of Source Code

Class Name	Class Name	Class Name
ASTActualPar	ASTLine	DynamicGen
ASTActualParList	ASTMyId	JJTTtcnParserState
ASTAttach	ASTOtherwise	Node
ASTBehaviourDescription	ASTPass	ParseException
ASTBehaviourLine	ASTRecv	SimpleCharStream
ASTCancelTimer	ASTRootTree	SimpleNode
ASTConsRef	ASTSend	Token
ASTConstruct	ASTStartTimer	TokenMgrError
ASTFail	ASTStatementLine	TtcnParser
ASTGoTo	ASTTestCase	LookaheadSuccess [private final]
ASTInconc	ASTTestCaseId	TtcnParserConstants
ASTIndent	ASTTestStepId	TtcnParserTokenManage
ASTLabel	ASTTreeReference	TtcnParserTreeConstants
ASTLabelId	ASTValue	TtcnParserVisitor

5.1.3 Reflexion Model tool file

We defined mapping rules in the Reflexion Model tool file, named “TestGen.rmt,” which contains the high-level model and the map that would be used in computing the Reflexion Model. We made pairs between classes and nodes of High Level Model in the mapping rules, and then we saved the file in the **parent folder** of source code. Table 2 shows “TestGen.rmt” file.

Table 2. TestGen.rmt File

```
<rmt>
<hlm>
<arc from="SymTab" to="Parser"/>
<arc from="Parser" to="SymTab"/>
<arc from="Parser" to="AST"/>
<arc from="Semantic" to="SymTab"/>
<arc from="Semantic" to="AST"/>
```

```
<arc from="AST" to="Semantic"/>
<arc from="AST" to="CodeGen"/>
<arc from="CodeGen" to="AST"/>
</hlm>
<map>
<entry class="Token" mapTo="Parser"/>
<entry class="SimpleCharStream" mapTo="Parser"/>
<entry class="TokenMgrError" mapTo="Parser"/>
<entry class="TtcnParser" mapTo="Parser"/>
<entry class="ParseException" mapTo="Parser"/>
<entry class="TtcnParserTreeConstants" mapTo="Parser"/>
<entry class="TtcnParserConstants" mapTo="Parser"/>
<entry class="TtcnTokenManager" mapTo="Parser"/>
<entry class="Node" mapTo="AST"/>
<entry class="SimpleNode" mapTo="AST"/>
<entry class="AST*" mapTo="AST"/>
<entry class="JJTTtcnParserState" mapTo="AST"/>
<entry class="TtcnParserVisitor" mapTo="CodeGen"/>
<entry class="DynamicGen" mapTo="CodeGen"/>
</map>
</rmt>
```

5.1.4 Reflexion Model

Reflexion Model is computed automatically whenever we select one menu, “Compute Reflexion Model.” We selected the Reflexion Model tool (.rmt) file in the Package Explorer of the Eclipse. Next, we loaded the pop-up menu by pressing the right button of the mouse. After that, we selected **Reflexion Model Tool > Compute Reflexion Model**. Finally, the Reflexion Model is computed and then displayed on the Reflexion Model view as Figure 3.

In the Reflexion view, our first Reflexion Model has 2 Convergences, 6 Divergences, and 6 Absences. A Convergence presented as a solid line indicates interactions discovered in the source that were expected by the developer in the high-level model. A Divergence presented as a dashed line indicates discovered interactions that were not expected by the developer. An absence presented as a dotted line indicates interactions that were expected but not found.

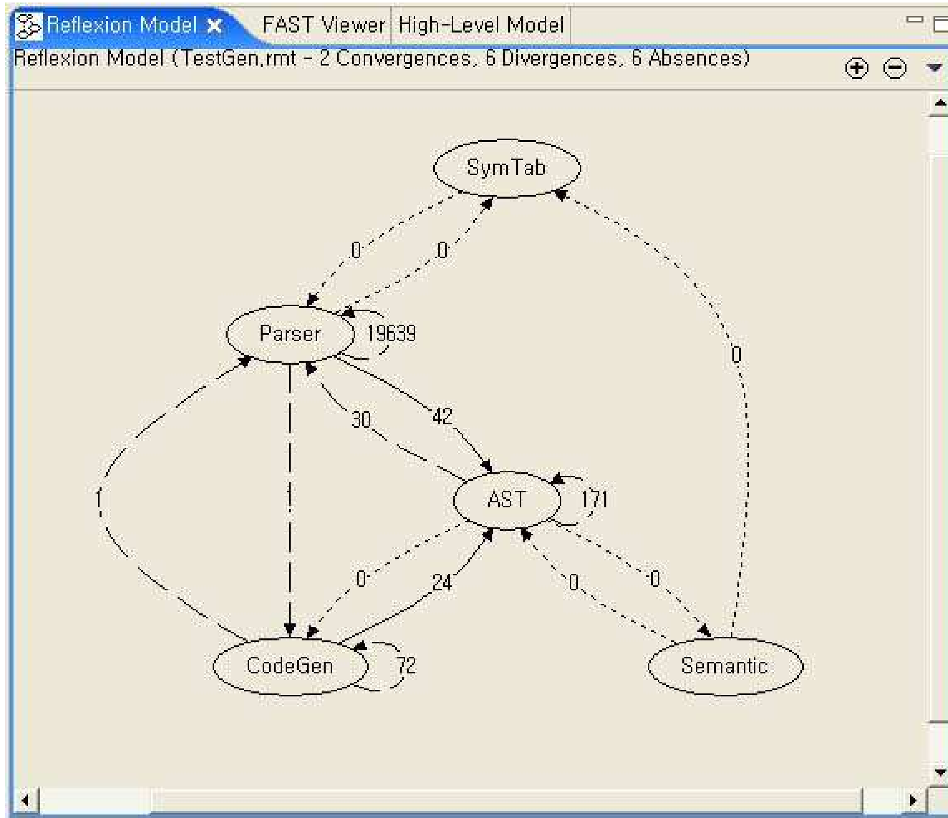

Figure 3. Reflexion Model

We learned that we should refine the parser node and delete semantic node after first computation of Reflexion Model. The number attached to each arc indicates the number of calls in the source associated with the interaction. The software Reflexion Model shown above summarizes 19978 calls found in the TestGen Java source. Parser has too many self-directed arcs, which means that the node should be divided to lexer and syntax node. Currently, our source code doesn't have semantic analysis part, we don't need to include in high-level model.

5.1.5 Reflexion Model Information view

This view shows the source model values corresponding to an arc in a Reflexion Model when we click on the arc in the Reflexion Model from which we expect to get more information on. The next Figure shows the information about the Reflexion Model. This figure shows the information about the method calls that occurs at the Parser node.

Javadoc | Reflexion Model Information | Reflexion Model Unmapped Values | Declaration | Problems | Tasks

Reflexion Model Information (from Parser to Parser - 19639 Occurences)

package (from)	class (from)	f...	method (from)	package (to)	class (to)	field (to)	method (to)
parser	TtcnParser		jj_3_226()	parser	TtcnParser		jj_3R_182()
parser	TtcnParser		jj_3_31()	parser	TtcnParser		jj_scan_token(int)
parser	TtcnParser		jj_3_31()	parser	TtcnParserCo...	SELECT...	
parser	TtcnParser		TCompDclsGroup()	parser	TtcnParser		jj_consume_token(int)
parser	TtcnParser		TCompDclsGroup()	parser	TtcnParser		TCompDclsGroupId()
parser	TtcnParser		TCompDclsGroup()	parser	TtcnParser		TCompDclsOrGroup()
parser	TtcnParser		TCompDclsGroup()	parser	TtcnParser		jj_2_364(int)
parser	TtcnParser		TCompDclsGroup()	parser	TtcnParser		jj_consume_token(int)
parser	TtcnParser		jj_3_504()	parser	TtcnParser		jj_3R_347()
parser	TtcnParser		jj_3R_321()	parser	TtcnParser		jj_scan_token(int)
parser	TtcnParser		jj_3R_321()	parser	TtcnParserCo...	LEFTBR...	
parser	TtcnParser		jj_3R_321()	parser	TtcnParser		jj_3R_323()
parser	TtcnParser		jj_3R_267()	parser	TtcnParser		jj_scan_token(int)
parser	TtcnParser		jj_3R_267()	parser	TtcnParser		jj_3R_525()
parser	TtcnParser		jj_2_596(int)	parser	TtcnParser	jj_la	
parser	TtcnParser		jj_2_596(int)	parser	TtcnParser	jj_lastpos	
parser	TtcnParser		jj_2_596(int)	parser	TtcnParser	jj_scanpos	
parser	TtcnParser		jj_2_596(int)	parser	TtcnParser	token	
parser	TtcnParser		jj_2_596(int)	parser	TtcnParser		jj_3_596()
parser	TtcnParser		jj_2_596(int)	parser	TtcnParser		jj_save(int,int)
parser	TtcnParser		jj_2_114(int)	parser	TtcnParser	jj_la	
parser	TtcnParser		jj_2_114(int)	parser	TtcnParser	jj_lastpos	
parser	TtcnParser		jj_2_114(int)	parser	TtcnParser	jj_scanpos	
parser	TtcnParser		jj_2_114(int)	parser	TtcnParser	token	
parser	TtcnParser		jj_2_114(int)	parser	TtcnParser		jj_3_114()
parser	TtcnParser		jj_2_114(int)	parser	TtcnParser		jj_save(int,int)
parser	TtcnParser		jj_3_728()	parser	TtcnParser		jj_3R_436()
parser	TtcnParser		jj_3R_545()	parser	TtcnParser		jj_scan_token(int)
parser	TtcnParser		CPs_Used()	parser	TtcnParser		jj_consume_token(int)
parser	TtcnParser		CPs_Used()	parser	TtcnParser		jj_2_391(int)
parser	TtcnParser		CPs_Used()	parser	TtcnParser		CP_List()

Figure 4. Reflexion Model Information view

5.1.6 Reflexion Model Unmapped Values

Our first model has 216 unmapped values. These are almost Java API libraries. If we don't specify map information for specific files, then the methods which these files have will be displayed on the unmapped value view.

Javadoc | Reflexion Model Information | Reflexion Model Unmapped Values | Declaration | Problems | Tasks

Reflexion Model Unmapped Values (216 Occurences)

package (from)	class (from)	field (from)	method (from)
java.lang	Object		
		length	
		length	
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	System		arraycopy(Object,int,Object,int,int)
java.lang	Error		
java.lang	Throwable		getMessage()
java.io	PrintStream		println(String)
java.lang	System	out	
java.lang	StringBuffer		
java.lang	StringBuffer		append(String)
java.lang	StringBuffer		toString()
java.io	PrintStream		println(String)
java.lang	System	out	
java.lang	Object		
java.io	PrintStream		print(String)

Figure 5. Reflexion Model Unmapped Values

5.2 The result: the second trial

5.2.1 High Level Model

We changed the High Level Model in Figure 2 to the High Level Model in Figure 6, which reflects the result of the Reflexion Model in the first trial. First, we eliminated the components that are connected with other components in absence relationship, because the components are the parts expected but not implemented. Second, we added two components to the High Level model in order to refine the Parser that has 19978 calls: TokenManager and Scanner.

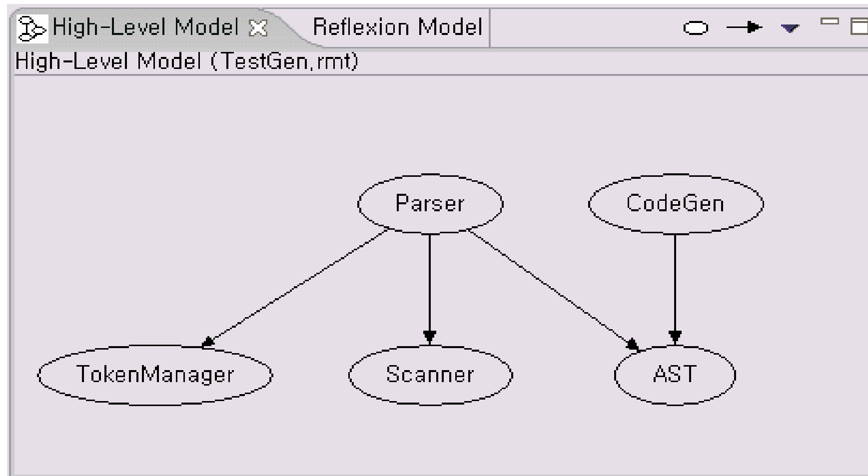

Figure 6. High Level Model

5.2.2 Reflexion Model tool file

We modified mappings between the components of source code and nodes of High Level Model in the mapping rules.

Table 3. The changes of the mapping rules

Entity type	Entity name	Node name
method	jjt*	AST
method	jj_*	Scanner
class	Token	TokenManager
class	SimpleCharStream	TokenManager
class	TtcnTokenManager	TokenManager
class	TokenMgrError	TokenManager

We added the method mapping into the mapping rules so as to divide Parser, the one class into several parts. We found that jj* methods are used to tokenize the file stream, so we separate the methods from the Parser and named the group as Scanner. After that, we noticed that there are some methods that should not be included in the Scanner but AST. Thus, we added another mapping rule for put the methods

in AST and changed the first rules from jj* to jj_*. Table 4 shows the mapping file containing the result.

Table 4. TestGen.rmt File

```
<rmt>
<hlm>
<arc from="Parser" to="AST"/>
<arc from="Parser" to="TokenManager"/>
<arc from="Parser" to="Scanner"/>
<arc from="CodeGen" to="AST"/>
</hlm>
<map>
<entry class="DynamicGen" mapTo="CodeGen"/>
<entry class="TtcnParserVisitor" mapTo="CodeGen"/>
<entry method="jjt*" mapTo="AST"/>
<entry method="jj_*" mapTo="Scanner"/>
<entry class="Token" mapTo="TokenManager"/>
<entry class="SimpleCharStream" mapTo="TokenManager"/>
<entry class="TtcnTokenManager" mapTo="TokenManager"/>
<entry class="TokenMgrError" mapTo="TokenManager"/>
<entry class="TtcnParser" mapTo="Parser"/>
<entry class="ParseException" mapTo="Parser"/>
<entry class="TtcnParserTreeConstants" mapTo="Parser"/>
<entry class="TtcnParserConstants" mapTo="Parser"/>
<entry class="Node" mapTo="AST"/>
<entry class="SimpleNode" mapTo="AST"/>
<entry class="AST*" mapTo="AST"/>
<entry class="JJTTtcnParserState" mapTo="AST"/>
</map>
</rmt>
```

5.2.3 Reflexion Model

In the Reflexion Model view, our second Reflexion Model has 4 Convergences, 11 Divergences, and 0 Absence. We refined Parser into three modules: Parser, Scanner and TokenManager. We noticed that the TokenManager is easily taken apart from the Parser in that the TokenManager is just called by Parser 37 times while it has 2597 internal calls; it shows that the TokenManager is high cohesive and low coupled. The Scanner is evaluated as valuable in that the separation from the Parser shows some meaning to us;

the Scanner has 4742 internal calls and called by Parser 4229 times and calls the Parser 4721 times. However, there is still a problem that the Scanner is highly coupled with the Parser.

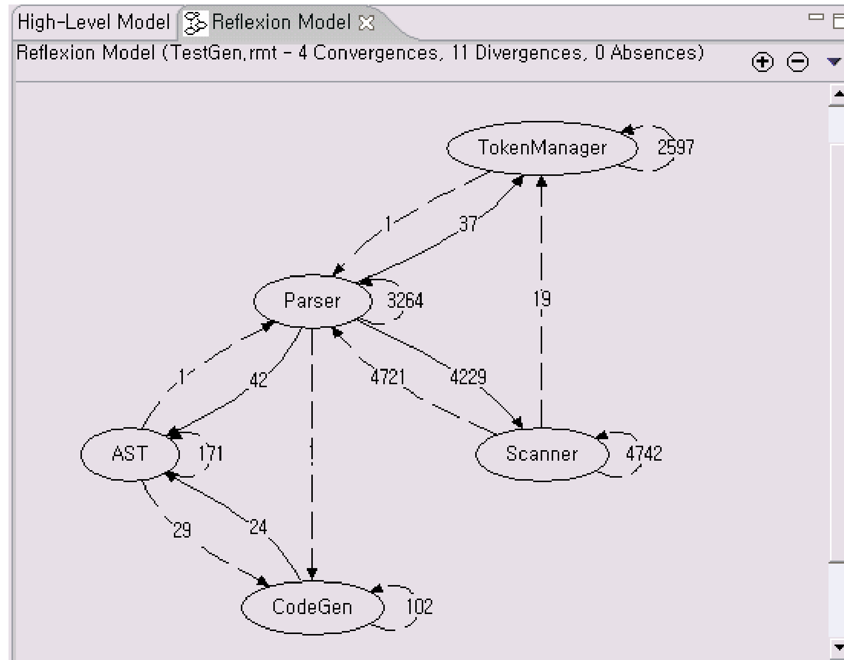

Figure 7. Reflexion Model

5.3 The third refinement

Simply viewing a displayed Reflexion Model does not generally provide sufficiently detailed information for a user to assess, plan, and perform a software engineering task. So, we used Reflexion Model Information to refine our High Level Model. We could find the need of refinement by investigating the Reflexion Model information because the information in the Reflexion Model may reveal missing interactions in the high-level model or deficiencies in the map.

The second point of refinement is unmapped value investigation. If we don't specify map information for specific class, method and field, then the unmapped values will be displayed on the unmapped value view. So, unmapped values are useful to refine the model. Our second model has 216 unmapped values. These are almost Java API libraries. So we didn't care the unmapped values since the second refinement.

5.3.1 High Level model

First, we changed the name of two high level nodes

- AST → Tree
- Scanner → Lexer

Second, we deleted TokenManager node because TokenManager is a part of Lexer node. Third, we add two new nodes: SymbolTable and Headers. Finally we modified the interaction between the nodes to

reflect the computed Reflexion Model.

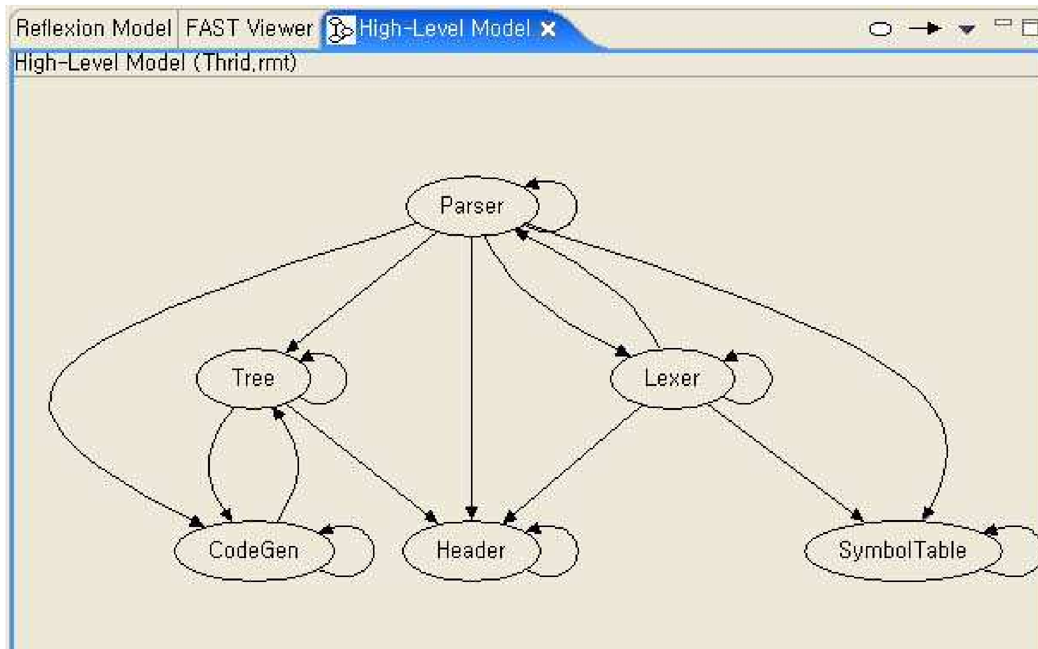

Figure 8. High Level Model

5.3.2 Reflexion Model tool file

We modified mappings between the source component and nodes of High Level Model in the mapping rules.

Table 5. The changes of the mapping rules

Entity type	Entity name	Node name
class	TtcnParserConstants	Header
class	TtcnParserTreeConstants	Header
method	jj_scan*	Lexer
field	jj_scan*	Lexer
field	jjto*	Lexer
method	jjt*	Tree
field	jjt*	Tree
class	SimpleCharStream	Lexer
class	TtcnParserTokenManager	Lexer
class	TokenMgrError	Lexer
class	Token	SymbolTable

We rearranged the order of mapping rules because reflexion tool can not support exact name mapping. For example next mapping order should be kept because TtcnParserTokenManager include Token string.

If the order is reversed, then TtcnParserTokenManager is mapped to SymbolTable.

```
<entry class="TtcnParserTokenManager" mapTo="Lexer"/>
<entry class="Token" mapTo="SymbolTable"/>
```

Table 6. TestGen.rmt File

```
<rmt>
<hlm>
<arc from="Parser" to="Tree"/>
<arc from="Parser" to="SymbolTable"/>
<arc from="Parser" to="Lexer"/>
<arc from="Parser" to="Header"/>
<arc from="Parser" to="Parser"/>
<arc from="Parser" to="CodeGen"/>
<arc from="Tree" to="Tree"/>
<arc from="Tree" to="CodeGen"/>
<arc from="Tree" to="Header"/>
<arc from="SymbolTable" to="SymbolTable"/>
<arc from="Lexer" to="Header"/>
<arc from="Lexer" to="Parser"/>
<arc from="Lexer" to="SymbolTable"/>
<arc from="Lexer" to="Lexer"/>
<arc from="Header" to="Header"/>
<arc from="CodeGen" to="Tree"/>
<arc from="CodeGen" to="CodeGen"/>
</hlm>
<map>
<entry class="TtcnParserConstants" mapTo="Header"/>
<entry class="TtcnParserTreeConstants" mapTo="Header"/>
<entry class="DynamicGen" mapTo="CodeGen"/>
<entry class="TtcnParserVisitor" mapTo="CodeGen"/>
<entry method="jj_scan*" mapTo="Lexer"/>
<entry field="jj_scan*" mapTo="Lexer"/>
<entry field="jjto*" mapTo="Lexer"/>
<entry method="jjt*" mapTo="Tree"/>
<entry field="jjt*" mapTo="Tree"/>
```

```
<entry class="SimpleCharStream" mapTo="Lexer"/>
<entry class="TtcnParserTokenManager" mapTo="Lexer"/>
<entry class="TokenMgrError" mapTo="Lexer"/>
<entry class="ParseException" mapTo="Parser"/>
<entry class="Node" mapTo="Tree"/>
<entry class="SimpleNode" mapTo="Tree"/>
<entry class="AST*" mapTo="Tree"/>
<entry class="JJTTtcnParserState" mapTo="Tree"/>
<entry class="TtcnParser" mapTo="Parser"/>
<entry class="Token" mapTo="SymbolTable"/>
</map>
<config>
</config>
</rmt>
```

5.3.3 Reflexion Model

The third Reflexion Model is computed and then displayed on the Reflexion Model view as Figure 3. The third Reflexion Model doesn't have any divergence and absence because we modified High Level Model to reflect the computed Reflexion Model.

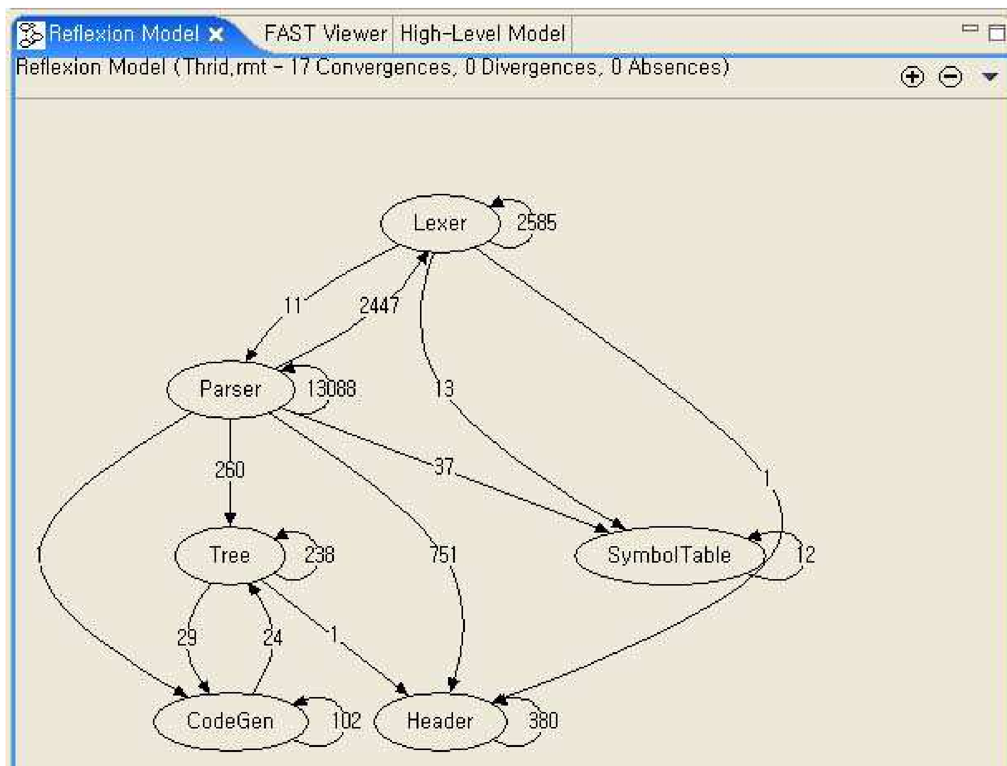

Figure 9. Reflexion Model

6. Evaluation on Reflexion Model

Advantage: A good thing of this tool is that a user does not need to extract any information from source by using extraction tool. When the number of source file is not so many (we have forty java files), we can map the source to the nodes of high level model without source model.

Disadvantage: The manual and explanation of the tool are not sufficient enough to run the tool properly. Specially, there is no technical follow-up when the versions of Eclipse and Java are upgraded. We had to find appropriate version of Eclipse and Java by attempting several trials.

6.1 Trial and errors

We failed 9 times before setting up successfully. The main cause of the failures was due to the version conflicts between JDK and Reflexion Model Tool. We summarized the failures and the reason in the Table 7.

Table 7. The Failures in using the Reflexion Model Tool

Num	Failure	Reason
1	Eclipse version: the tool did not create *.rmt file	The Eclipse plug-in version does not work properly with JDK 1.5.0
2	Eclipse version: the menu and view of reflection model was not shown when downloading the software and installing it	The local plug-in update does not work
3	Eclipse version: the menu and view of reflection model was not shown even if the eclipse version was changed to 2.2.1	Only the remote plug-in update does work
4	Standalone version: the tool did not show the GUI window	The tool requires another software, Graphviz
5	Standalone version: the tool shows an error message	The standalone version works with JDK 1.2.2
6	Eclipse version: the tool was not installed	The tool requires Draw2d plug-in software
7	Eclipse version: the high level model was not drawn	The help manual has no concrete guide in drawing the model
8	Eclipse version: the reflection model was not generated	The .rmt file must be in the java project to be analyzed

9	Eclipse version: the reflection model was not generated	The file mapping mechanism does not work

6.2 Evaluation statistics

We experimented 3 times in order to get the expected model. The total time was 71 hours. The average time of our task per each trial was 8 hours It shows the different result from the lecture of the analysis class in the Apr. 28. According the lecture, the task of defining the model takes about 15~ 60 minutes and the task of defining the mapping file takes about 10~30 minutes. Thus, the time seems to take too much when compared to the time in the lecture. However, we needed the time for discussion and experimentation for several models for acquiring each expected model. Thus, we guess that the subject might know well the Reflexion Model Tool or the structure of the target source code.

Table 8. The time consumed in each activity

Activity	Hours
Experimental Setup	48 hours
The fist trial	7 hours
The second trial	7 hours
The third trial	9 hours
Total	71 hours

Table 9. The number of lines of the mapping specification

Activity	The number of lines of the specification
The fist trial	30
The second trial	26
The third trial	44

Table 10. The errors of the tool itself

Num	Description
1	Once either the toolbar of ‘Add Nodes’ or ‘Add Arcs’ is chosen, one of the two is always in a selected state.
2	Te file mapping mechanism in the paper does not work
3	The mapping rules should be ordered from the long name of the file to the short name of the file.

7. Additional Experiment

7.1 Experiment Setting

The original intention of the selection of this tool is extracting an architecture from the source code generated by JavaCC. During the extraction of the architecture, we were curious about the capability of this tool. In addition to that, we were wondering about this tool can extract an architecture from a set of source code in terms of reverse engineering. It's because we got to know the fact that this tool only shows the relationships among nodes, which are defined in a High Level Model, based on the method invocation and the use of fields. So, we decided to experiment this tool can identify implicit invocations.

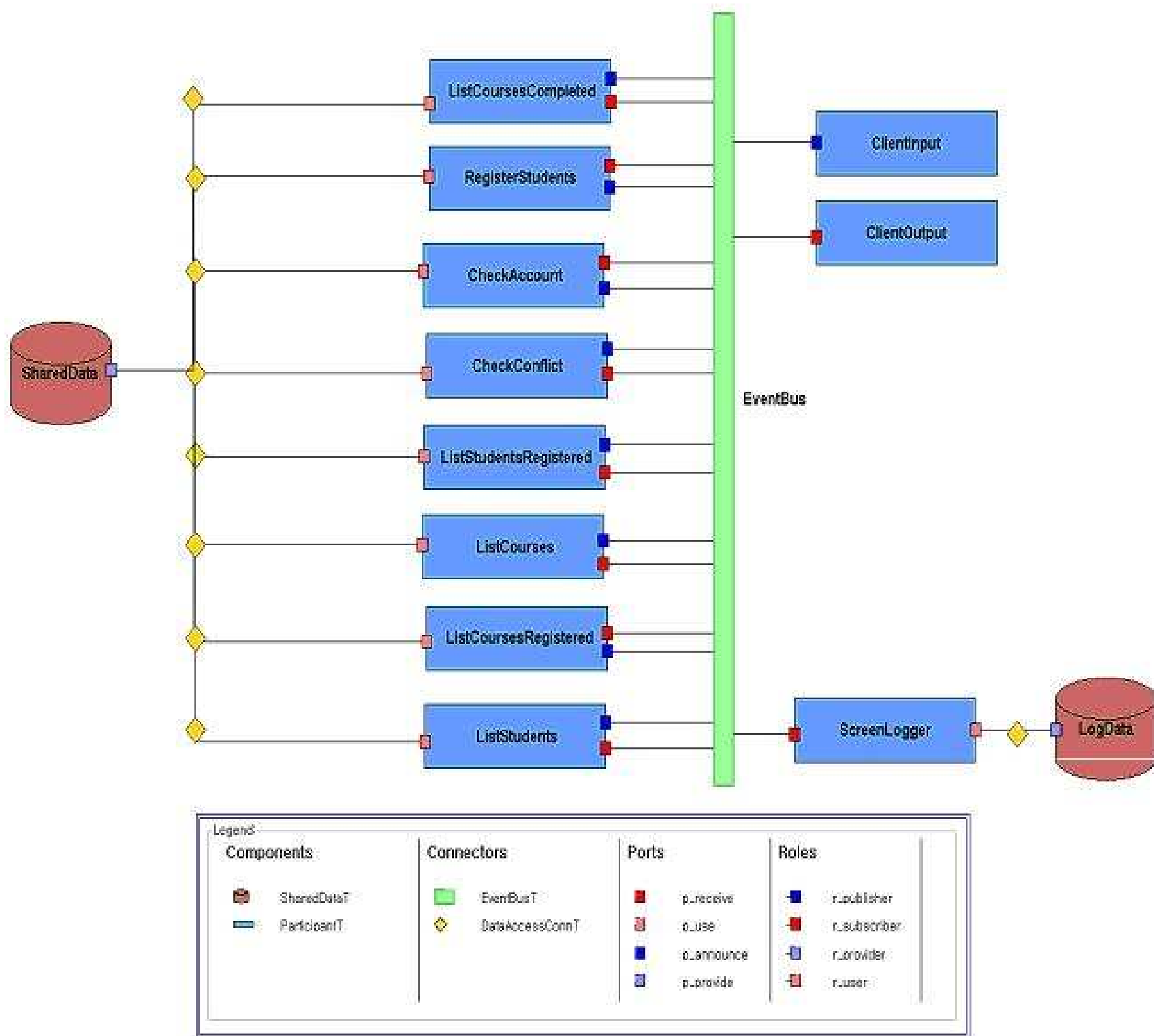

Figure 10 Implicit Invocation Style Architecture Style of a System

The way to experiment is specifying a High Level Model based on an architecture, and then extracting the relationships among the nodes that are defined in the High Level Model. We selected a system that has been developed for an assignment of the architecture course. The system is architected based on implicit invocation style. The architecture is like the following:

All the components, which are attached on EventBus, are interacting with each other based on the implicit invocation. And, except for ClientInput and ClientOutput, components are interacting with the shared data components including LogData and SharedData based on explicit invocation.

Then, we specified a High Level Model for the above architecture as the following. One is the textual specification of the High Level Model and the other is the graphical specification of the High Level Model.

```
<rmt>
<hlm>
<arc from="ListAllCourse" to="EventBus"/>
<arc from="ListAllStudents" to="EventBus"/>
<arc from="ListCoursesCompleted" to="EventBus"/>
<arc from="ListCoursesRegistered" to="EventBus"/>
<arc from="ListStudentsRegistered" to="EventBus"/>
<arc from="RegisterStudent" to="EventBus"/>
<arc from="CommandEvent" to="EventBus"/>
<arc from="ConflictCheck" to="EventBus"/>
<arc from="AccountCheckHandler" to="EventBus"/>

<arc from="EventBus" to="ListAllCourse"/>
<arc from="EventBus" to="ListAllStudents" />
<arc from="EventBus" to="ListCoursesCompleted" />
<arc from="EventBus" to="ListCoursesRegistered" />
<arc from="EventBus" to="ListStudentsRegistered" />
<arc from="EventBus" to="RegisterStudent" />
<arc from="EventBus" to="CommandEvent" />
<arc from="EventBus" to="ConflictCheck" />
<arc from="EventBus" to="AccountCheckHandler" />

<arc from="EventBus" to="ClientOutput"/>
<arc from="ClientInput" to="EventBus"/>
```

```
<arc from="EventBus" to="ScreenLogger"/>

<arc from="ListAllCourse" to="DataBase"/>
<arc from="ListAllStudents" to="DataBase"/>
<arc from="ListCoursesCompleted" to="DataBase"/>
<arc from="ListCoursesRegistered" to="DataBase"/>
<arc from="ListStudentsRegistered" to="DataBase"/>
<arc from="RegisterStudent" to="DataBase"/>
<arc from="CommandEvent" to="DataBase"/>
<arc from="ConflictCheck" to="DataBase"/>
<arc from="AccountCheck" to="DataBase"/>

<arc from="DataBase" to="ListAllCourse" />
<arc from="DataBase" to="ListAllStudents" />
<arc from="DataBase" to="ListCoursesCompleted" />
<arc from="DataBase" to="ListCoursesRegistered" />
<arc from="DataBase" to="ListStudentsRegistered" />
<arc from="DataBase" to="RegisterStudent" />
<arc from="DataBase" to="CommandEvent" />
<arc from="DataBase" to="ConflictCheck" />
<arc from="DataBase" to="AccountCheck" />

</hlm>
<map>
<entry class="ListAllCourseHandler" mapTo="ListAllCourse"/>
<entry class="ListAllStudentsHandler" mapTo="ListAllStudents" />
<entry class="ListCoursesCompletedHandler" mapTo="ListCoursesCompleted"/>
<entry class="ListCoursesRegisteredHandler" mapTo="ListCoursesRegistered" />
<entry class="ListStudentsRegisteredHandler" mapTo="ListStudentsRegistered"/>
<entry class="RegisterStudentHandler" mapTo="RegisterStudent" />
<entry class="CommandEventHandler" mapTo="CommandEvent" />
<entry class="ConflictCheckHandler" mapTo="ConflictCheck" />
<entry class="AccountCheckHandler" mapTo="AccountCheck" />
</map>
</rmt>
```

Table 11 High Level Model of Architecture

We modeled the High Level Model so that a node in the model can match with a component in the architecture diagram. In addition to that, a component that announces an event is modeled in the way that the relationship a node for the component between a node for EventBus is pointed to the EventBus node from the component node. In the case of the component that receives an event, the relationship a node for the component between a node for EventBus is pointed to the component node from the EventBus node.

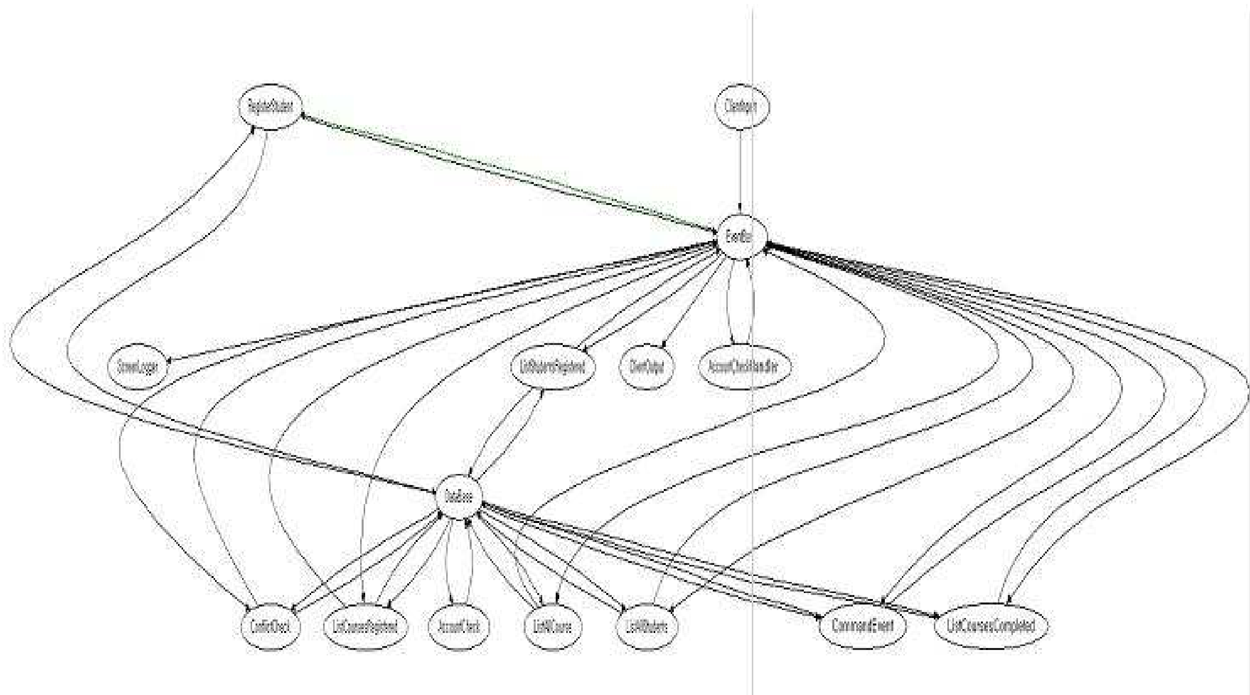

Figure 11 High Level Model of Architecture

7.2 Experimental Result

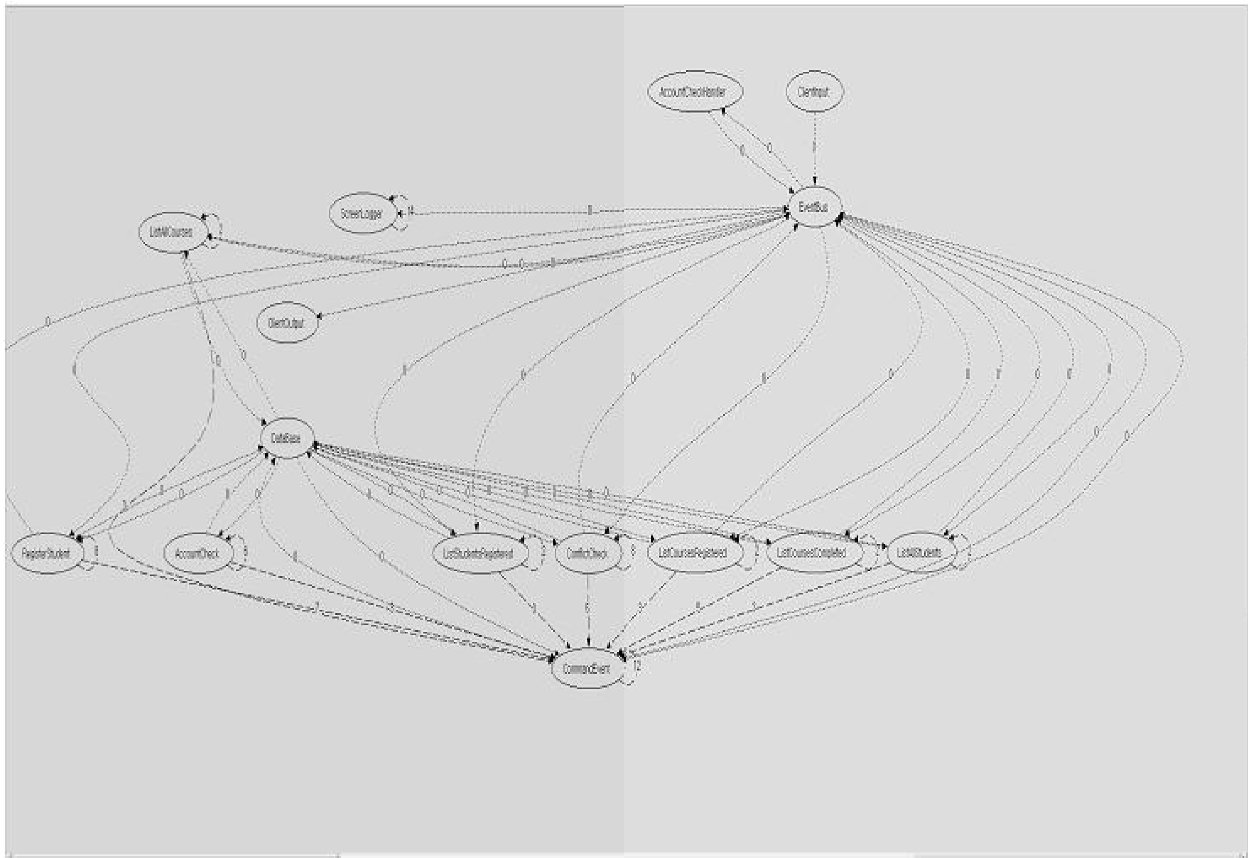

Figure 12 Reflexion Model as a result

The results gave us a shock, because there was no convergence, even there are several components that explicitly invoke SharedData or LogFile. However, the tool found 39 absences and 15 divergences. One interesting fact is that the tool didn't find the relationship between a component that explicitly invokes SharedData or LogFile. It's because the tool cannot identify the inheritance structure among the source code. For example, ListAllCourseHandler that is inherited from CommandEventHandler refers to its objDataBase. However, the Reflexion tool considered these references as references to objDataBase of CommandEventHandler.

```
abstract public class CommandEventHandler implements Observer {

    protected DataBase objDataBase;


    protected int iOutputEvCode;


    public CommandEventHandler(DataBase objDataBase, int iCommandEvCode, int iOutputEvCode) {
        // Subscribe to command event.
        EventBus.subscribeTo(iCommandEvCode, this);

        // Remember the database reference and output event name.
        this.objDataBase = objDataBase;
        this.iOutputEvCode = iOutputEvCode;
    }


    public void update(Observable event, Object param) {
        // Announce a new output event with the execution result.
        EventBus.announce(this.iOutputEvCode, this.execute((String) param));
    }


    abstract protected String execute(String param);
}
```

Figure 13 CommandEventHandler.java

```
public class ListAllCoursesHandler extends CommandEventHandler {


    public ListAllCoursesHandler(DataBase objDataBase, int iCommandEvCode, int iOutputEvCode) {
        super(objDataBase, iCommandEvCode, iOutputEvCode);
    }

    protected String execute(String param) {
        // Get all course records.
        ArrayList vCourse = this.objDataBase.getAllCourseRecords();

        // Construct a list of course information and return it.
        String sReturn = "";
        for (int i=0; i<vCourse.size(); i++) {
            sReturn += (i == 0 ? "" : "\n") + ((Course) vCourse.get(i)).toString();
        }
        return sReturn;
    }
}
```

Figure 14 ListAllCoursesHandler.java

Problems | Declaration | Search | High-Level Model | Reflexion Model | Reflexion Model Information | Reflexion Model Unmapped Values | Tasks | Hierarchy

Reflexion Model Information (from ListAllCourses to CommandEvent - 3 Occurences)

package (from)	class (from)	field (from)	method (from)	package (to)	class (to)	field (to)	method (to)
arch	ListAllCoursesHandler		execute(String)	arch	CommandEventHandler	objDataBase	
arch	ListAllCoursesHandler		execute(String)	arch	CommandEventHandler	objDataBase	
arch	ListAllCoursesHandler			arch	CommandEventHandler		

Figure 15 Reflexion Model Information

7.3 Experimental Summary

- Inappropriateness for extracting architecture

  In architecture world, the style that architecture follows is important, because the style says how a component executes its functionalities and how components interacts with each other. From this experiment, because the Reflexion Model tool doesn't know the information about how components interact with each other, the tool couldn't catch the implicit invocations. So, in order for the tool to be able to extract an architecture from source code, the tool should provide a way to make it know the architectural information.

- No support of analyzing inheritance structure

  In objected oriented programming, the concept of inheritance is very important, because the concept helps a developer to make a system modular. However, the fact that the tool is not able to analyze the inheritance structure makes it very difficult to guarantee the output of the tool for a system that can be implemented using the inheritance concept.

8. Possible Improvements

8.1 Supporting hierarchical High Level Modeling

The Reflexion Model could be more useful when supporting hierarchical modeling. We used the tool in analyzing one portion of the tool, Parser generated by JavaCC. After investigating the first result, we eliminated the other parts that could be valuable for stakeholders' understanding but unnecessary for this analysis. This elimination helped us to grasp the internal of the Parser, because maintaining a simple structure let us concentrate on delving into the Parser. On the other hand, the High Level Model and Reflexion Model as the final result do not look understandable for our stakeholders, because the external modules of the parser have been eliminated. If the tool supports hierarchical modeling, we could leave the module that we had to remove and the hierarchical High Level Model and Reflexion Model would be helpful for the people who try to understand the final result without the history of the analysis.

8.2 Looking into the target code

The Reflexion Model could be helpful for developer to look into the target code if the tool provides the capability of addressing the source code when we double-click a method (or field) in the Reflexion Model information. The tool shows two views for each line in the Reflexion Model after computation: Reflexion Model Information and Reflexion Model Unmapped Values. Two views contain the list of methods. The

views let us know which methods belong to the relationships between two components, and which methods are not included in any component. Nonetheless, we did not understand the roles of the methods when we reviewed the information. Thus, we had to study the internal code of the specific method and we used the search function of Eclipse in order to find the method shown in the view. If the tool provides the feature that locates the position of the code when a user double-clicks the method in the list of the view, the user feels more convenient in using the tool.

8.3 Auto-generation of first High Level Model

The Reflexion Model may provide the option to auto-generate the first High Level Model by analyzing the cohesion and coupling among the modules of source code. Reflexion Model has two assumptions. The first is that an engineer knows the high level structure of the source code approximately, so the tool requires High Level Model identified by the engineer at the first time. The second is that the source code is not monolithic, so the tool shows the convergence, divergence and absence information of nodes after its computation. The assumptions of the tool address the common situation in development projects, so the usefulness of the tool can be acceded. On the other hand, some developers having no knowledge and involving the project at the first time may not know the high level structure of the system. In that case, the first high level provided by the Reflexion Model tool could be beneficial for the engineer.

8.4 Using meta model to describe the architectural styles

The Reflexion Model could be helpful for developers to re-engineer a system, like the case of this project, if it provides two functions. One is to provide a way to describe architectural styles and the other is to map the description into the reverse engineering. A meta model can be used to describe the architectural styles. A meta model contains general information about the architecture. Our suggestion is that when Reflexion Model reengineers a system, a developer can select one of predefined styles. So, by repeatedly applying a style at a time and reengineering a system, the developer can extract an architecture more exactly.

9. Conclusion

We partitioned the source code seeming like a black box of our product into lexical analyzer, syntax analyzer, symbol table, tree and header and grasped the relationships among them. The understanding helped us to mitigate the risks that might occur when the task of restructuring the product is required in that we already have the knowledge needed to modify the software product. This tool may be used or may be not in the summer semester because the mapping information shown by the tool could be used in re-

factoring the source code according to the architecture in our minds. Re-factoring the source code could be required or not; if we implement our software product conforming our architecture well, we would not require the use of the tool. Yet, if we are unsure whether our source code complies with our architecture, then we would use the Reflexion Model tool.

The Reflexion Model tool was useful because the result facilitated the propagation of the domain expert's knowledge. Our team members had different level of knowledge of compiler structure. One of our team members was conversant with a compiler development, and he did not feel the usefulness of the tool. Another team member had no knowledge of a compiler and had some difficulty in defining a High Level Model by himself, so he insist that the tool was not useful in capturing some architecture from the source. Other three team members were somewhat familiar with the structure of the compiler, but did not know the internal structure of the parser. They did understand the internal structure by manipulating the High Level Model, reviewing the Reflexion Model and discussing with the domain expert. The Reflexion Model helped us to share the same picture of the internal structure of the Parser.

In conclusion, even though the Reflexion Model is not appropriate for re-engineering some kinds of system mentioned in chapter 7, it is meaningful in that they opened a direction to re-engineering. Furthermore, the tool seems to motivate our team to make progresses of state of the arts, by using the knowledge that we learned from the Analysis course.

Reference

[1] G.C. Murphy and D. Notkin, "Reengineering with Reflexion Models: A Case Study", IEEE Computer 30, 8, 1997, pp.29-36.
[2] G.C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models", In the Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995, ACM, New York, NY, p. 18-28.
[3] G.C. Murphy, Reflexion Models, http://www.cs.ubc.ca/~murphy/jRMTool/doc/, December 2003