Static Analysis of Lyee Requirements for Legacy System Software

Hamido Fujita and Mohameed Mejri

Abstract— ***Software development has been suffering, for many decades, from the lack of simple and powerful methodologies and tools. Despite the tremendous advances in this research field, the crisis has still not been overcome and the proposed remedies are far from resolving the problems of software development and maintenance, Lately, a new and very promising methodology, called Lyee, has been proposed. It aims to automatically generate programs from simple user requirements.***

The purpose of this paper is, on the one hand, to provide a short and technical introduction to the Lyee software development methodology, and on the other hand, to show how some classical static analysis techniques (execution time and memory space optimization, typing, slicing, etc.) can considerably improve many aspects of this new methodology. This paper contributes on introducing new techniques for software system design, for legacy systems.

Index Terms—*Software development, Lyee methodology, static analysis, dynamic analysis*

1. INTRODUCTION

SOFTWARE development and maintenance have become activities at major importance to our economy. As software comes into widespread use, this activity involves a large industry. Hundreds of billions of dollars are spent every year in order to develop and maintain software.

Manuscript received May 4, 2004.

Prof. H. Fujita is with the Faculty of Software and Information Science, Iwate Prefectural University, Iwate, 020-0193 Japan (E-mail: issam@soft.iwate-pu.ac.jp)

Prof. M. Mejri is with the Faculty of Software and Information Science, Department of Computer Science and Software Engineering, Laval University, Quebec, G1K 7P4, Canada

Today, competition between actors is the software development field is fiercer than ever. To remain in the race, these actors (companies) must keep productivity at its peak and costs low. They must also deliver products (software) at high quality and deliver them on time. However, an important question is whether the available tools and methodologies for software development suit company needs.

The work reported here contributed to research study on building legacy systems based on Lyee frame work, (http://www.lyee-project.soft.iwate-pu.ac.jp).

More detailed version of this paper is available at http://www.lyee-project.soft.iwate-pu.ac.jp/IPSI .

Basically, the goal of software development research is to find ways how to build better software easily and quickly. A large variety of methodologies and techniques have been proposed and elaborated on, over the last 10 years, to improve one or more steps in the software development life cycle. Despite their considerable contributions, they have had difficulty to finding their way into widespread use. In fact, almost all of them fail to produce clearly understood and modifiable systems and their use is still considered to be an activity accessible only to specialists with a large array of competencies, skills, and knowledge. This, in turn, leads to highly paid personal, high maintenance costs, and extensive checks needing to be performed on the software.

Lyee [1-4] (governementaL methodologY for softwarE providencE) is specific new and promising methodology. Intended to deal efficiently with a wide range of software problems related to different fields, Lyee allows the development of software by

simply defining its requirements. More precisely, a developer has only to provide words, calculation formulae, calculation conditions (preconditions) and layout of screens and printouts, and then leaves in the hands of the computer all subsequent troublesome programming process (e.g., control logic aspects). Despite its recency, the results of the use of Lyee have shown tremendous potential. In fact, compared to conventional methodologies, development time, maintenance time and documentation volume can be considerably reduced by using Lyee (as much as 70 to 80\%) **2**. Up to now, a primitive supporting tool called **LyeeAll2** has been available to developers allowing the automatic generation of code from requirements. Nevertheless, as with any new methodology, further research is needed on Lyee to investigate its efficiency, to discover and eliminate its drawbacks, and to improve its good qualities.

In this paper, we show how classical static analysis techniques can considerably contribute the analysis of Lyee requirements (a set of words within their definitions, their calculation conditions and their attributes) in order to help their users understand them, discover their inconsistencies and incomplete or erroneous parts, and generate codes of better qualities (consuming less memory and execution time). Basically, the static analysis techniques we investigate are:

? Optimization techniques (constant propagation, communication sub expression detection, etc.) to generate better Lyee programs.
? Slicing techniques to abstract requirements to their relevant part needed for some analysis.
? Typing techniques to automatically generate types and to discover typing errors.

The remainder of this paper is organized as follows. In Section 2, we give a short and technical introduction to the Lyee methodology. Section 3 shows how static analysis techniques can contribute to the enhancement of this methodology. Section 4 introduces LyeeAnalyzer, a prototype that we have developed to implement some static analysis techniques. Finally, Section 5 provides concluding remarks on this work, and discusses future research.

2. THE LYEE METHODOLOGY

Most people who have been seriously engaged in the study and development of software systems agree that one of the most problematic tasks in this process is that of understanding requirements and correctly transforming them. To solve this problem, the Lyee methodology proposes a simple method for to generating programs from requirements.

With the Lyee methodology, requirements are given as a set of statements containing words together with their definitions, their calculation conditions and their attributes (input/output, type, etc.). A word is an atomic element and its definition and calculation conditions show its interaction wit the other words.

Although the philosophic principles behind the Lyee methodology are interesting, in this section we focus only on some practical ideas useful to understand how to write software using this methodology and how to understand the code that is automatically generated from Lyee requirements.

2.1 Lyee requirements

Within the Lyee methodology, requirements are given in a declarative way as a set of *statements* containing words together with their definitions, their calculation conditions and their attributes (input/output, types, security attributes, etc.). For the sake of simplicity, throughout this paper, we consider each statement to contain the following information:

? Word: An identifier of a word.
? Definition: An expression defining a word.

We suppose, for the sake of simplicity, that an expression can be one of the following: (where _ is the empty expression)

Exp := *val* | *id* _ (*Exp*) | op Exp | Exp op Exp

val:= *num* | *num*.*num* | *bool*

num:= *0*| *1*.. |*9*| *num num*

bool:= *true*| *false*

id:=*a*| ….|*z*| *A*| …. |*Z*| *id num*| *id id*

Op :=+| -|*|*or* |*and* |< |<= |= |<> |> |>= |*not*

? Condition: the calculation condition of the word; this is an expression *Exp* that must be boolean. If there is no condition (the condition is always true) we leave this field empty.

? IO: specifies whether the defined word is an input, output or intermediate word (neither an input nor an output). If the word is an input, this field can take the value IF (input form file) or IS (input from screen). Similarly, if the word is an output, then this field can take the value OF or OS. However, if the word is intermediate, we leave this field empty.

? Type: specifies the type of the word. It can take on one of values *int*, *float* or *bool*.

? Security: associates a security level with the defined word. It takes on one of the following values *public* or *secret*.

Notice that the fields "Type" and "Security" can be empty if the defined word is not an input. Notice also, that other types and other security levels can be easily incorporated to support others Lyee requirements.

In the rest of this paper, if *s* is a statement, then we use:

? s_w
to denote the statement defining a word *w*.

? Definition (*s*)
to denote the field "Word" of *s*.

? Condition(*s*)
to denote the field "Condition" of s.

? IO(*s*)
to denote the field "IO" of *s*.

? Type(*s*)
to denote the field "Type" of *s*.

? Security(*s*)
to denote the field "Security" of *s*.

Table 1 below gives an example of Lyee requirements.

Word	Definit-ion	Condit-ion	IO	Type	Securi-ty
a	b+c	b*e>2	OF	int	secret
c			IS	float	public
b	2*c+5	c>0		float	public
e				float	public

Table 1: Example of Lyee requirements.

The requirements given in Table 1, correspond intuitively, in a traditional programming language, to the code given in Table 2.

Stat e-ment	Code
s_a	If b*e>2 then a:=b+c; output(a); endif
s_c	Input(c);
s_b	If c>2 then b:=2*c+5; output(b); endif
s_e	Input(e);

Table 2: Statement code

Within the Lyee methodology, the user does not need to specify the order (control logic) in which these definitions will be executed. As shown in Table 2, despite the fact that the definition of word a uses word b, statement s_b is listed after the statement s_a. As explained in the sequel, form these requirements, and independent of the order of statements, Lyee is able to generate code that computes all the defined words.

This simple idea has, as shown in [1-4], multiple beneficial consequences on the different steps of software development. In fact it allows us to begin developing of software even with incomplete requirements.

Moreover, the user need not deal with control logic as with more classical methodologies. The control logic part of the software will be, within the Lyee methodology, automatically generated reducing consequent programming errors and time.

Flexibility is also a major benefit of the Lyee methodology since the maintenance task can be reduced to a simple modification of requirements (add, remove and/or modify words' definitions).

2.2 Code Generation

From the requirements in Table 1, we can automatically generate a program that computes the values of a and b and outputs them. This program will simply repeat the execution of these instructions until a fixed point is reached, i.e., any other iteration will not change the value of any word as shown in Fig. 1

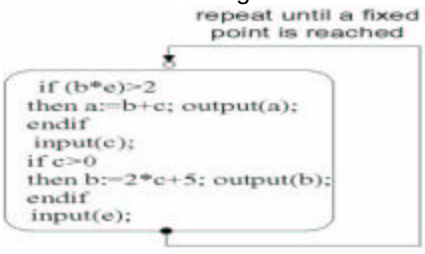

Fig. 1, Requirement Execution.

Let's be more precise about the structure and the content of the program that will be automatically generated by Lyee from requirements. Within the Lyee methodology, the execution of a set of statements, such as the ones given in Table 1, is accomplished in a particular manner. In fact, Lyee distributes the code associated with statements over three spaces, called *Pallets* (*W02*, *W03* and *W04*) in the Lyee terminology, as shown in Fig.2.

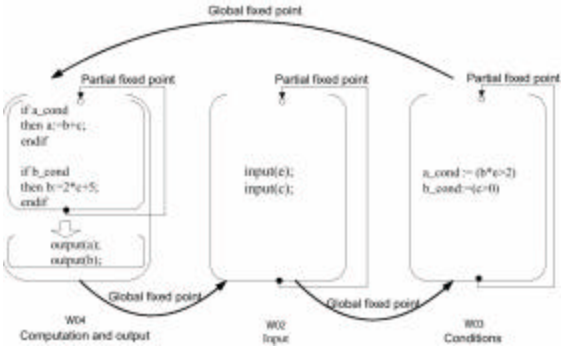

Fig.2. Lyee Pallets

Pallet *W02* deals with the input words. Pallet *W03* computes the calculation conditions of the words and saves the results in boolean variables. For instance, the condition "b*e>2" used within the definition of the word "a" is calculated in *W03* and the true/false result is saved in another variable "a_cond". Finally, pallet *W04* deals with the calculation of the words according to their definition given within the requirements. It also, outputs the values of the computed words.

Starting from pallet *W04*, a Lyee program tries to compute the values of all the defined words until a fixed point is reached. Once there is no evolution in *W04* concerning the computation of the word values, control is given to pallet *W02*. In its turn, this second pallet tries repeatedly to input values of words until a fixed point is reached (no others inputs are available) and then transfer the control to pallet *W03*. Finally, and similar to pallet *W04,* pallet *W03* tries to compute the calculation conditions of the words according to the requirements until a fixed point is reached. As shown in Fig.3, this whole process (W04 ? W03 ? W02) will repeat until a situation of overall stability is reached and the three pallet linked together are called Scenario Function.

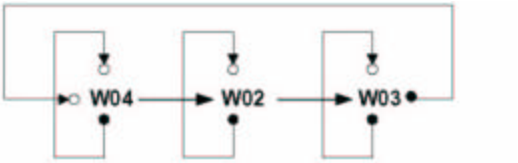

Fig.3. Scenario Function

Fig.4. Predicate Vector

In addition, Lyee has established a simple program with a fixed structure (called a Predicate Vector in the Lyee terminology) that makes the structure of generated code uniform and independent of the requirement content. The global program will be simple calls of predicate vectors. The structure of a predicate vector is as shown in Fig.4.

The goal of a predicate vector changes from one pallet to another. For instance, in the pallet W04, the first goal is to give a value to a word according to its definition. For the example shown in Fig. 2, the predicate vectors associated with the calculation of the word "a" and the word "b" are as shown in Fig.5.

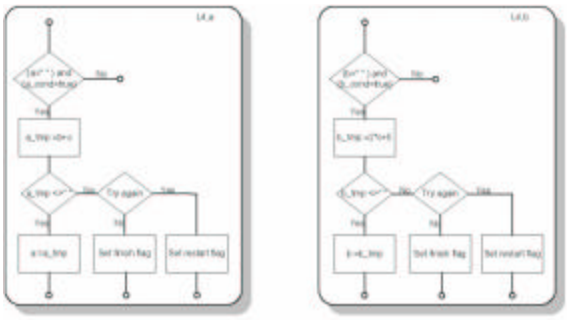

Fig. 5. The Predicate Vectors of *L4*, *a* and *L4*, *b*.

Finally, in pallet *W03*, the goal of the predicate vectors is to compute preconditions specified within requirements as shown in Fig. 6.

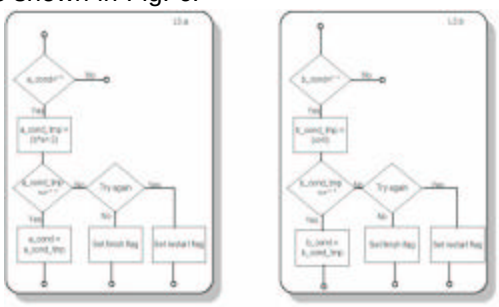

Fig.6. The predicate Vectors of *L3*, a and *L3*,b.

Finally, the Lyee program associated with the requirements given in Table 1 is as shown in Table 3.

pallet	Program	Comments
W04	Call S4 Do Call L4_a Call L4_b While a fixed point is not reached Call 04 Call R4	Initialize memory Calculate a Calculate b Output the result Go to W02
W02	Do Call L2_e Call L2_c While a fixed point is not reached Call I2	
	Call R2	
W03	Do Call L3_a Call L3_b While a fixed point is not reached Call R3	 Calculate a_cond Calculate b_cond Go to W04

Table: 3 Lyee Generated Program

2.3 Process Route Diagrams

The Scenario Function presented in the previous section can be a complete program for a simple case of given requirements, particularly when all the input and output words belong to the same screen and there is no use of any database. However, if we need to input and output words that belong to databases or to different screens interconnected together, then the situation will be more complicated. For the sake of simplicity, we deal, in the sequel, only with the case when we have many screens. Suppose for instance that we have three interconnected screens, as shown in Fig.7, allowing a user to navigate from one to another.

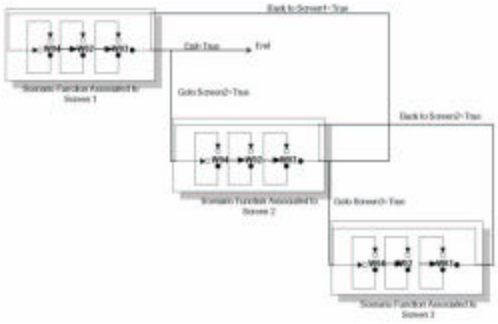

Fig. 7. Screen Interactions

In fact, some screens may not be visited for a given execution of the program and then the computation of the value of their words will be lost. For that reason, Lyee associates with each screen its owner scenario function that will be executed only if this screen is visited. The scenario functions associated with screens are connected together showing the move from one of them to another. In the Lyee terminology, many scenario functions connected together make up a Process Route Diagram as shown in Fig. 8.

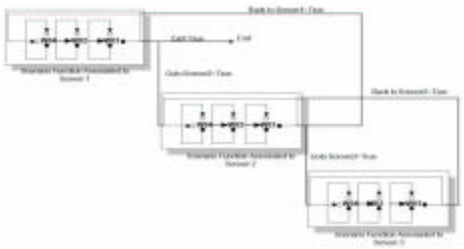

Fig. 8. Processes Route Diagram

To summarize, according to the Lyee methodology, a program usually contains several PRDs. Each of these is a set of interconnected scenario functions and each scenario function contains three interconnected pallets *W02*, *W03* and *W04*.

2.4 Drawback of the Methodology

In spite of the Lyee methodology is simplicity and its several positive impacts on all the steps of the software development cycle, it suffers from a major drawback, namely, the size of the generated code. In fact, to each word given within requirements, it attributes several memory areas. For more details about the exact amount of memory consumed by each word, the reader can refer to [2,3].

The remainder of this paper shows how static analysis techniques can help produce Lyee programs that run faster and consume less memory space, as well providing other beneficial qualities.

3. STATIC ANALYSIS OF LYEE REQUIREMENTS

Software static analysis [5,6] generally means the examination of the code of a program without running it. Experience has shown that many quality attributes of specifications and codes can be controlled and improved by static analysis techniques. In particular, static analysis techniques can make programs run faster and use less memory, and they can help locate faults. Applied on requirements, static analysis finds logic errors and omissions before the code is, generated and consequently allows the user to save precious development and testing time. The purpose of this section is to pinpoint some static analysis techniques that could improve the qualities of Lyee requirements and code generated from those requirements.

4. LYEEANALYZER

The LyeeAnalyzer prototype was developed to demonstrate the static analysis techniques presented in this paper.

4.1 Inputs and Outputs

The LyeeAnalyzer takes as input Lyee requirements and can produce as output slices and ordered requirements suitable for the generation of optimized code by the LyeeAll2 tool. In addition, it can perform other requirement optimizations, such as constant propagation, and verifications such as type safety. The interface to of the prototype is as shown in Figure.9. The buttons in the top part of the window propose to access to the different static analysis techniques implemented in the tool; the inputs defined on the left frame, and the out in the right hand frame.

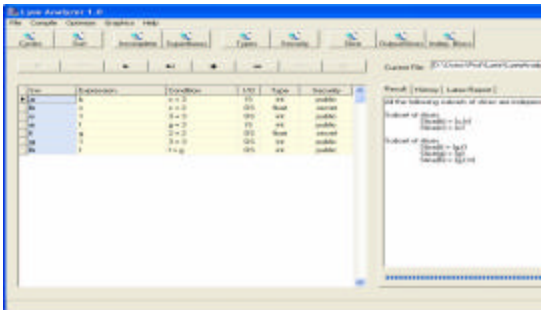

Fig.9. LyeeAnalyzer main interface

4.2 LyeeAnalyzer Architecture

The basic components of this prototype are the following:

? Lexical and Syntactic Analyzers: These components take as input Lyee requirements and give as output a syntactic tree commonly called an intermediate representation. This new representation of requirements is the starting point of all the static analysis techniques that we perform. Furthermore, when parsing the Lyee requirements, lexical or syntactic errors can be detected and communicated to the user.

? Semantics Analyzer: This component allows the discovery of type errors, security violations, incomplete statements, dead statements, cyclic statements and superfluous statements. It also allows us to generate the missing types, i.e., it generates the types of all output and intermediate words.

? Flow-Based Analyzer: Starting from the intermediate representation generated by the previous components, the flow-based analysis component generates all information related to the circulation of control and data flow from one requirement point to another. The results of these analyses consist of two graphs:
?Control Flow Graph: Each node of this graph contains a single statement of requirements and an edge between two nodes represents direct flow of control between them.
? Data-Flow Graph: Each node of this graph contains a single statement of requirements and an edge between two nodes represents a data flow (Def/Use information) between them.
?Optimizer: Amongst others, this component implements the constant propagation techniques and generates an ordered and simplified sequence of statements suitable for the LyeeAll2 tool to produce a program that runs faster and consumes less memory.
? Slicer: This component takes as input flow information (such as the Def/Use information associated with each word) generated by the Flow-Based Analysis component and one or many slicing criteria and gives as output slices that correspond to these criteria. Within the classical programming language, a slicing criterion is generally considered a pair C=(s,V), where s is a statement and V a set of variables. Slicing a program according to this criterion means the generation of all statements relevant to the computation of the variables in V given before the statement s. The LyeeAnalyzer uses the VCG (Visualization of Compiler Graphs)[13] tool to display the various involved results (independent bloc, optimized code, etc.).
The LyeeAnalyzer prototype is intended to achieve at least the following goals:
? Help the LyeeAll2 tool to generate efficient programs.
? Help the user to understand and maintain Lyee requirements, especially for those containing a large number of statements: Among others, the slicing technique is potentially suitable for this goal.
? Help the user debug requirements: Finding incomplete or inconsistent requirements.
? Automatic parallelization: Identifies independent slices that could be computed in parallel.
? Automatic generation of types: Given the types of the input words, this tool generates the types of output and intermediate words.

5. CONCLUSION AND FUTURE WORK

We have reported in this paper the use of static analysis techniques on the Lyee requirements and their impacts. First, we have shown how classical optimization techniques such as constant propagation and common subexpression detection can be used to improve the execution time of the Lyee programs. We have also shown how to discover errors in requirements (dead definition, cyclic definition, incomplete or superfluous definitions). Second, we have shown how slicing techniques can potentially improve the understanding and the maintenance of Lyee systems. Also, we have shown how to find independent parts of Lyee systems that can be executed in parallel using slicing techniques. Third, we have proposed a type system allowing both the detection of typing errors and the automatic generation of types of the intermediate and output words. Forth, we have shown how the Lyee methodology is suitable for some extensions such as security aspects. Some of the presented static analysis techniques are now implemented in a prototype called LyeeAnalyzer. As future work, we want to investigate other static and dynamic analysis techniques to improve other aspects of the Lyee methodology.

ACKNOWLEDGMENT

Special thanks are due to Dr Fumio Negoro, all the members of his group and those of Catena Co., for their assistance and comments and suggestions.

REFERENCES

[1] F. Negoro, Principle of Lyee software, 2000 International Conference on Information Society in 21st Century (IS2000) (2000) 121–189.

[2] F. Negoro, Introduction to Lyee, The Institute of Computer Based Software Methodology and Technology, Tokyo, Japan, 2001.

[3] F. Negoro, I. Hamid, A proposal for intention engineering, 5th East-European Conference

Advances in Databases and Information System (ADBIS'2001).
[4] F. Negoro, I. Hamid, A proposal for intention engineering, International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2001) .
[5] M. Bozga, J. C. Fernandez, L. Ghirvu, Using static analysis to improve automatic test generation, 2000, pp. 235–250.
[6] S. Muchnick, Compiler Design Implantation, Morgan Kaufman Publishers, California, 1999.
[7] T. HENNING, Optimization Methods, Springer-Verlag, 1975.
[8] W. Weiser, Program slicing, IEEE Trans Software Eng. (1984) 352–357.
[9] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.
[10] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security 4 (3) (1996) 167–187.

Hamido Fujita;
Professor at Faculty of Software and Information Science, at Iwate Prefectural University, Iwate, JAPAN, also, he is the director of Intelligent System Software Laboratory, and Director of Cognitive thinking Systems Laboratory, both at Iwate Prefectural University. Also, Prof. Fujita is the main leader of Lyee International Research Project,
http://www.lyee-project.soft.iwate-pu.ac.jp/
He is also the general Chair of the SOMET conferences series,http://www.lyee-project.soft.iwate-pu.ac.jp/en/conference/index.html.