Program Mining Augmented with Empirical Properties

Minh Ngoc Ngo
Nanyang Technological University, Singapore

Hee Beng Kuan Tan
Nanyang Technological University, Singapore

INTRODUCTION

Due to the need to reengineer and migrating aging software and legacy systems, reverse engineering has started to receive some attention. It has now been established as an area in software engineering to understand the software structure, to recover or extract design and features from programs mainly from source code. The inference of design and feature from codes has close similarity with data mining that extracts and infers information from data. In view of their similarity, reverse engineering from program codes can be called as program mining. Traditionally, the latter has been mainly based on invariant properties and heuristics rules. Recently, empirical properties have been introduced to augment the existing methods. This article summarizes some of the work in this area.

BACKGROUND

"Software must evolve over time or it becomes useless" (Lehman). A large part of software engineering effort today is involved not in producing code from scratch but rather in maintaining and building upon existing code. For business, much of their applications and data reside in large, legacy systems. Unfortunately, these systems are poorly documented. Typically, they become more complex and difficult to understand over time.

Due to this need to reengineer and migrating aging software and *legacy systems*, reverse engineering has started to receive some attentions. *Reverse engineering* is an approach to understand the software structure, to recover or extract design and features, given the source code. This process identifies software building blocks, extract structural dependencies, produces higher-level abstractions and present pertinent summaries. Reverse engineering helps by providing computer assistance, often using compiler technologies such as lexical, syntactic and semantic analysis. *Static analysis* strengthens the study by inferring relationships that may not be obvious from the syntax of the code, without running the program.

The inference of *design patterns* and features from codes has close similarity with data mining that extracts and infers information from data. In view of their similarity, reverse engineering from program codes can be called as program mining. Program mining is a challenging task because there are intrinsic difficulties in performing the mapping between the language of high level design requirements and the details of love level implementation. Although program mining depends heavily on human effort, a number of approaches have been proposed to automate or partially automate this process.

Traditionally, approaches to program mining are based on invariant properties of programs or heuristic rules to recover design information from source code (De Lucia, Deufemia, Gravino, & Risi, 2007; Poshyvanyk, Gueheneuc, Marcus, Antoniol, & Rajlich, 2007; Robillard & Murphy, 2007; Shepherd, Fry, Hill, Pollock, & Vijay-Shanker, 2007; Shi & Olsson, 2006). Recently, several researchers have developed *experimental program analysis* approaches (Ruthruff, Elbaum, & Rothermel, 2006; Tonella, Torchiano, Du Bois, & Systa, 2007) as a new paradigm for solving software engineering problems where traditional approaches have not succeeded. The use of empirical properties, which have been validated statistically, to solve problems is very common in the area of medicine (Basili, 1996). However, the application of this method is rather unexplored in software engineering. This paper summarizes some of our work in this area which incorporates the use of empirical properties.

MAIN FOCUS

In this section, we first describe our work on empirical-based recovery and maintenance of input error correction *features* in information system (Ngo & Tan, 2006). We then discuss the use of empirical properties to infer the infeasibility of a program path (Ngo & Tan, 2007). This work on infeasible path detection is useful for all software engineering tasks which rely on static analysis especially testing and coverage analysis. Finally, we present an approach to extract all the possible database interactions from source code (Ngo, Tan, & Trinh, 2006).

Empirical Recovery and Maintenance of Input Error Correction Features

Information systems constitute one of the largest and most important software domains in the world. In many information systems, a major and important component is processing inputs submitted from its external environment to update its database. However, many input errors are only detected after the completion of execution. We refer to this type of input errors as after-effect input errors. As after-effect input error is unavoidable, the provision of after-effect input error correction features is extremely important in any information system. Any omission of the provision of these features will lead to serious adverse impact. We observe that input error correction features exhibits some common properties. Through realizing these properties from source code, input error correction features provided by a system to correct these effects can be recovered. All the empirical properties have been validated statistically with samples collected from a wide range of information systems. The validation gives evidence that all the empirical properties hold for more than 99 percent of all the cases at 0.5 level of significance.

Properties of Input Error Correction Features

Input errors can be approximately classified into: error of commission (EC), error of omission (EO) and value error (VE). In a program, there are statements which when being executed will raise some external effects; these statements are called effect statements. If effect statements are influenced by some input errors, they will result in erroneous effects; we refer to these as effect errors. Effect errors can be classified into EO, EC and VE in the combination of attributes for executing the effect statement.

In an information system, a type of effect error (EO, EC or VE) that may occur in executing an effect statement e can be corrected by executing another effect statement f in the system. We call f an error correction statement (Tan & Thein, 2004) for correcting e. The minimum collection of all the paths for correcting an input error ξ is called the basis collection of error correction paths for correcting ξ. We discover some empirical patterns for realizing the basis collection of error correction paths for correcting an input error as follow:

Empirical Property 1. A set of paths $\{q_1,\ldots, q_k\}$ is a basis collection of error correction paths for correcting the input ξ is and only if we can partition the set of effect errors resulting from ξ into several partitions $\{E_1,\ldots, E_k\}$ such that for each j, $1 \le j \le k$, by executing q_j, all the effect errors in E_j are corrected.

For many programs, the correctability of all its input errors can be deduced from existence of basis collection of error correction paths for some of these errors. This is presented in the following empirical property:

Empirical Property 2. If for each path through the control flow graph of a program, there is a basis collection of error correction paths for correcting EC in the input accessed in the path, then any after-effect input error of the program is correctable.

If each path through the control flow graph of a program S is in a basis collection of error correction paths for correcting an input error of program T, then S is called an error correction program for T. The following empirical property suggests a mechanism to verify the correctability of a program V.

Empirical Property 3. It is highly probable that any input error of program V is correctable if and only if one of the following conditions holds:

- Based on basis collections of error correction paths, Empirical Property 2 infers that any after-effect error of V is correctable.
- Program V is an error correction program for program T and any after-effect error of T is correctable.