Chapter 13
Mining Past-Time Temporal Rules
A Dynamic Analysis Approach*

David Lo
Singapore Management University, Singapore

Siau-Cheng Khoo
National University of Singapore, Singapore

Chao Liu
Microsoft Research – Redmond, USA

ABSTRACT

Specification mining is a process of extracting specifications, often from program execution traces. These specifications can in turn be used to aid program understanding, monitoring and verification. There are a number of dynamic-analysis-based specification mining tools in the literature, however none so far extract past time temporal expressions in the form of rules stating: "whenever a series of events occur, previously another series of events happened before". Rules of this format are commonly found in practice and useful for various purposes. Most rule-based specification mining tools only mine future-time temporal expression. Many past-time temporal rules like "whenever a resource is used, it was allocated before" are asymmetric as the other direction does not holds. Hence, there is a need to mine past-time temporal rules. In this chapter, the authors describe an approach to mine significant rules of the above format occurring above a certain statistical thresholds from program execution traces. The approach start from a set of traces, each being a sequence of events (i.e., method invocations) and resulting in a set of significant rules obeying minimum thresholds of support and confidence. A rule compaction mechanism is employed to reduce the number of reported rules significantly. Experiments on traces of JBoss Application Server and Jeti instant messaging application shows the utility of our approach in inferring interesting past-time temporal rules.

DOI: 10.4018/978-1-60566-758-4.ch013

INTRODUCTION

Different from many engineering products that rarely change, software changes often throughout its lifespan. This phenomenon has been well studied under the umbrella notion of software evolution. Software maintenance effort deals with the management of such changes, ensuring that the software remains correct while additional features are incorporated (Grubb & Takang, 2003). Maintenance cost can contribute up to 90% of software development cost (Erlikh, 2000). *Reducing maintenance cost* and ensuring a program *remains correct during evolution* are certainly two worthwhile goals to pursue.

A substantial portion of maintenance cost is due to the difficulty in understanding an existing code base. Studies show that program comprehension can contribute up to 50% of the maintenance cost (Fjeldstad & Hamlen, 1983; Standish, 1984). A challenge to software comprehension is the maintenance of an accurate and updated specification as program changes. As a study shows, documented specifications often remain unchanged during program evolution (Deelstra et al., 2004). One contributing factor is the short-time-to-market requirement of software products (Capilla & Duenas, 2003). Multiple cycles of software evolution can render the outdated specification invalid or even misguiding.

To ensure correctness of a software system, model checking (Clarke et al., 1999) has been proposed. It accepts a model and a set of formal properties to check. Unfortunately, difficulty in formulating a set of formal properties has been a barrier to its wide-spread adoption (Ammons et al., 2002). Adding software evolution to the equation, the verification process is further strained. First, ensuring correctness of software as changes are made is not a trivial task: a change in one part of a code might induce unwanted effects resulting in bugs in other parts of the code. Furthermore, as a system changes and features are added, there is a constant need to add new properties or modify outdated properties to render automated verification techniques effective in detecting bugs and ensuring the correctness of the system.

Addressing the above problems, there is a need for techniques to automatically reverse engineer or mine formal specifications from program. Recently, there has been a surge in software engineering research to adopt machine learning and statistical approaches to address these problems. One active area is specification discovery (Ammons et al., 2002; Cook & Wolf, 1998; Lo & Khoo, 2006; Reiss & Renieris, 2001), where software specification is reverse-engineered from program traces. Employing these techniques ensures specifications remain updated; also it provides a set of properties to verify via formal verification tools like model checking. To re-emphasize, the benefits of specification mining are as follows:

1. Aid program comprehension and maintenance by automatic recovery of program behavioral models, *e.g.*, (Cook & Wolf, 1998; Lo & Khoo, 2006; Reiss & Renieris, 2001).
2. Aid program verification (also runtime monitoring) in automating the process of "formulating specifications", *e.g.*, (Ammons et al., 2002; Yang et al., 2006).

Most specification miners extract specifications in the form of automata (Ammons et al., 2002; Cook & Wolf, 1998; Lo & Khoo, 2006; Reiss & Renieris, 2001) or temporal rules (Lo et al., 2008a; Yang et al., 2006). Usually a mined automaton expresses the whole behaviour of a system under analysis. Mined rules express strongly-observed constraints each expressing a property which holds with certain statistical significance.

Rules mined in (Lo et al., 2008a; Yang et al., 2006) express future-time temporal expressions. Yang et al. mine two event rules of the form: "Whenever an event occurs, eventually another event occurs in the future". Lo et al. mine temporal rules of arbitrary length of the form: "Whenever a