Mining Software Specifications

David Lo
National University of Singapore, Singapore

Siau-Cheng Khoo
National University of Singapore, Singapore

INTRODUCTION

Software is a ubiquitous component in our daily life. It ranges from large software systems like operating systems to small embedded systems like vending machines, both of which we frequently interact with. Reducing software related costs and ensuring correctness and dependability of software are certainly worthwhile goals to pursue.

Due to the short-time-to-market requirement imposed on many software projects, documented software specifications are often lacking, incomplete and outdated (Deelstra, Sinnema & Bosch 2004). Lack of documented software specifications contributes to difficulties in understanding existing systems. The latter is termed program comprehension and is estimated to contribute up to 45% of total software cost which goes to billions of dollars (Erlikh 2000, Standish 1984; Canfora & Cimitile 2002; BEA 2007). Lack of specifications also hampers automated effort of program verification and testing (Ammons, Bodik & Larus 2002).

One solution to address the above problems is mining (or automatic extraction of) software specification from program execution traces. Given a set of program traces, candidate partial specifications pertaining to the behavior a piece of software obeys can be mined.

In this chapter, we will describe recent studies on mining software specifications. Software specification mining has been one of the new directions in data mining (Lo, Khoo & Liu 2007a, Lo & Khoo 2007). Existing specification mining techniques can be categorized based on the form of specifications they mine. We will categorize and describe specification mining algorithms for mining five different target formalisms: Boolean expressions, automata (Hopcroft, Motwani & Ullman 2001), Linear Temporal Logic (Huth & Ryan 2003), frequent patterns (Han & Kamber 2006) and Live Sequence Charts (Harel & Marelly 2003).

BACKGROUND

Different from many other engineering products, software changes often during its lifespan (Lehman & Belady 1985). The process of making changes to a piece of software e.g., to fix bugs, to add features, etc., is known as software maintenance. During maintenance, there is a need to understand the current version of the software to be changed. This process is termed as program comprehension. Program comprehension is estimated to take up to 50% of software maintenance efforts which in turn is estimated to contribute up to 90% of total software costs (Erlikh 2000, Standish 1984; Canfora & Cimitile 2002). Considering the $216.0 billion of software component contribution to the US GDP at second quarter 2007, the cost associated with program comprehension potentially goes up to billions of dollars (BEA 2007). One of the root causes of this problem is the fact that documented software specification is often missing, incomplete or outdated (Deelstra, Sinnema & Bosch 2004). Mining software specifications is a promising solution to reduce software costs by reducing program comprehension efforts.

On another angle, software dependability is a well sought after goal. Ensuring software runs correctly at all times and identifying bugs are two major activities pertaining to dependability. Dependability is certainly an important issue as incorrect software has caused the loss of billions of dollars and even the loss of lives (NIST 2002; ESA & CNES 1996; GAO 1992). There are existing tools for performing program verification**.** These tools take formal specifications and automatically check them against programs to discover inconsistencies, identify bugs or ensure that all possible paths in the program satisfy the specification (Clarke, Grumberg & Peled 1999). However, programmers' reluctance and difficulty in writing formal specifications have been some of the barriers to the widespread adoption

of such tools in the industry (Ammons, Bodik & Larus 2002, Holtzmann 2002). Mining software specifications can help to improve software dependability by providing these formal specifications automatically to these tools.

MAIN FOCUS

There are a number of specification mining algorithms available. These algorithms can be categorized into families based on the target specification formalisms they mine. These include specification miners that mine Boolean expressions (Ernst, Cockrell, Griswold and Notkin 2001), automata (Cook & Wolf 1998; Reiss & Reinieris, 2001; Ammons, Bodik & Larus 2002; Lo & Khoo 2006a; Lo & Khoo 2006b; Mariani, Papagiannakis and Pezzè 2007; Archaya, Xie, Pei & Xu, 2007; etc.), Linear Temporal Logic expressions (Yang, et al. 2006; Lo, Khoo & Liu 2007b; Lo, Khoo & Liu 2008, etc.), frequent patterns (Li & Zhou, 2005; El-Ramly, Stroulia & Sorenson, 2002; Lo, Khoo & Liu 2007a; etc.) and Live Sequence Charts (Lo, Maoz & Khoo 2007a, Lo, Maoz & Khoo 2007b).

These mined specifications can aid programmers in understanding existing software systems. Also, a mined specification can be converted to run-time tests (Mariani, Papagiannakis & Pezzè 2007; Lo, Maoz & Khoo 2007a; Lo, Maoz & Khoo 2007b) or input as properties-to-verify to standard program verification tools (Yang, Evans, Bhardwaj, Bhat and Das, 2006; Lo, Khoo & Liu 2007b).

Preliminaries

Before proceeding further, let us describe some preliminaries. Specifications can be mined from either traces or code. A program trace is a sequence of events. Each event in a trace can correspond to a statement being executed, or a method being called, etc. In many work, an event is simply the signature of a method that is being called. Traces can be collected in various ways. A common method is to instrument a code by inserting `print' statement to various locations in the code. Running the instrumented code will produce a trace file which can then be analyzed.

Mining Boolean Expressions

Ernst, Cockrell, Griswold and Notkin (2001) propose an algorithm that mines Boolean expressions from program execution traces at specific program points. Sample Boolean expressions mined are x=y+z, x>5, etc. The algorithm is based on a set of templates which is then matched against the program execution traces. Template instances that are satisfied by the traces above a certain threshold are outputted to the user.

Mining Automata

Simply put, an automaton is a labeled transition system with start and end nodes. Traversing an automaton from start to end nodes will produce a sentence, which will correspond to a program behavior (e.g., file protocol: open-read-write-close). An automaton represents a set of valid sentences that a program can behave. An example of an automaton representing a file protocol is drawn in Figure 1.

One of the pioneering work on mining automata is the work by Ammons, Bodik and Larus (2002). In their work, a set of pre-processed traces are input to an automata learner (Raman & Patrick, 1997). The output of the learner is a specification in the form of an automaton learned from the trace file. This automaton is then presented to end users for fine tuning and modifications.

Lo and Khoo (2006a) define several metrics for assessing the quality of specification mining algorithms that mine automata. Among these metrics, precision and recall are introduced as measures of accuracy to existing specification miners producing automata. Precision refers to the proportion of sentences accepted by the language described by the mined automaton that are also accepted by the true specification. Recall refers to

Figure 1. File protocol specification