Addressing software bottlenecks: Amplifying human capabilities with tools

Suraj C. Kothari

Electrical and Computer Engineering Department, Iowa State University.

Email: kothari@iastate.edu Phone: (515) 294-7212

MOTIVATION

Software is a critical part of many modern technologies. We are at a point where we are constrained because of the human resources required to produce high quality software reliably. We lack efficient and reliable mechanisms for understanding and modifying software; these are critical bottlenecks in spite of advances in programming languages, development processes, and component-based methodologies.

Consider the problem of detecting and correcting defects such as memory leaks, invalid memory accesses, and security vulnerabilities. Since such defects are caused by unintended behaviors, we do not have specifications against which we can write test cases to catch such defects. Testing can easily miss the execution paths that contain the defects, and these defects can surface later and cause significant damages. Consider another problem--that of converting from sequential to parallel programs. This is known to be a very difficult problem that can consume significant human resources. There are many such examples, where even after investing significant human resources we may be unable to produce the desired result with existing complex large-scale software.

We need tools to automate tedious and mechanical tasks and to assist human experts in applying effective strategies to analyze and modify complex software. Similar to the medical profession, where medical instruments and surgical tools allow doctors to observe internals of complex body structures and make minimally invasive changes to correct the functioning of the body, software engineering should have tools to observe the internals of complex software and make fast and accurate software modifications to enhance functionality or correct defects.

STATIC ANALYSIS TOOLS

Static analysis is one method that has been developed to catch software defects in source code. Unlike a test run where only one execution path gets examined at one time, static analysis provides an efficient way to examine all execution paths at once. It does not require input values to force execution through various paths. Static analysis tools serve as an important extension of compilers. Compilers can catch relatively simple defects such as missing ends of parentheses. Static analysis tools can catch complex defects such as missing calls for deallocating memory. Static analysis tools perform global control and data flow analyses to catch complex defects. It is extremely time consuming to perform such analyses manually with large bodies of code. In minutes, a tool can perform an analysis that may take weeks to do manually. Moreover, manual analysis is tedious and thus prone to errors.

Static analysis has its own limitations. Static analysis techniques run into intractable computational complexity problems which make it difficult to scale the underlying data flow analysis to large software without sacrificing significant accuracy. For example, there are pointer analysis techniques that are fast but not very accurate because they are not flow and context sensitive. This inaccuracy may dramatically increase the number of false positives when searching for defects. Concurrent processing and runtime bindings are other barriers for static analysis. In the presence of concurrent processing, there is a possibility that matching memory allocation and deallocation calls may be executed in two different threads. If this is case, the static control flow analysis reaches a dead end, and we can get erroneous results.

AMPLIFYING HUMAN CAPABILITIES

Human experts can perform remarkably complex program analysis and transformation tasks which cannot be performed by static analysis tools. For example, human experts can parallelize very complex, large-scale codes. Human experts have an advantage in that they can make use of domain knowledge of the underlying numerical method while parallelizing a numerical modeling code. While we do not believe that it will be possible to build completely automated tools like compilers for many complex program analysis and transformation problems, we do believe that a new generation of tools can be built to greatly amplify human capabilities. Human experts can design good strategies based on their domain knowledge. However, it is difficult for them to connect their domain knowledge with specific implementation and to execute their strategies manually. Tools can assist them in executing their strategies and will allow them to achieve significant savings of time and effort as well as dramatic improvements in the quality of software.

PATTERN-BASED INTERACTIVE TOOLS

We conclude with some thoughts on how to build tools that can amplify human capabilities and lead to significant improvements in the productivity and quality of software. These ideas have evolved through our research projects on domain-specific software tools and subsequent commercialization of our technology.

We advocate pattern-based tools where the analysis and transformations performed by tools are driven by patterns that are defined by experts in a particular domain (e.g., security, safety, real-time, fault-tolerance, parallel processing). A pattern-based approach is a way to use domain knowledge in tools and avoid brute-force generic static analysis methods that lead to intractable problems. Formal mechanisms for specifying patterns can be: (a) a language for referencing program artifacts, and (b) a language for describing relationships between program artifacts. These formal mechanisms can serve as a useful framework for building a variety of tools for different programming languages and application domains.

We also advocate interactive tools where the user designs a strategy and uses the tool to execute that strategy efficiently and reliably. After the user decides the type of relationships he or she must look for in order to understand relevant aspects of the software (we will call this the “strategy”), the tool will extract all instances of the specified relationship and present information that the user can incorporate into a refined strategy. For example, a user may specify another relationship (a refined strategy) which will act as a filter and lead to a smaller and more useful subset of instances. Interactions also provide users a way of defining appropriate events and execution paths that the tool can incorporate when performing simulations of program behaviors. These controlled simulations can be faster and significantly more accurate than static analysis.

In summary, pattern-based interactive tools can be designed to employ domain knowledge in order to perform efficient analysis and transformation of software. They can automate tedious and time consuming mechanical tasks that are best performed by tools. This will enable effective use of human resources by allowing engineers to focus on designing effective strategies for understanding and modifying complex software.