Efficient SAT-Based Software Analysis: from Automated Testing to Automated Verification and Repair

Nazareno Aguirre
Departamento de Computación, FCEFQyN
Universidad Nacional de Río Cuarto, and
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
Email: naguirre@dc.exa.unrc.edu.ar

Formal approaches to software development have traditionally aimed at guaranteeing software correctness, through the use of notations, analysis mechanisms and other elements founded on solid mathematical grounds. Since the seminal works of Hoare, Floyd and others, formal methods have used logical notations to capture intended software behavior, and proposed techniques for reasoning about software correctness, originally mainly through deductive approaches. Formal methods for software development have been presented as a stronger correctness guarantee compared to informal, widely employed techniques such as testing. While advantages of formal methods make them appealing, the above-described original composition of formal methods implied two main difficulties for their effective use, namely, the need to dominate the logical notations used to specify intended software behavior, and mastering the associated deductive techniques necessary for analysis, i.e., for verifying software correctness.

The mentioned difficulties related to formal methods have caused many practitioners to consider them worthwhile mostly for safety-critical and other kinds of software whose correct behavior is crucial. With the advent of model checking, automated deduction and other *automated* formal analysis techniques, the second of the above mentioned difficulties became less important, since manual or semi-automated deductive techniques associated with the use of formal methods started to be replaced by fully automated ones. This (then) new trend in formal methods helped to broaden the adoption of formal methods in various software engineering contexts, but had its own problems, most notably *scalability* (automated analyses are often accompanied by a high computational cost) and what we will call *thoroughness* (automation generally requires analyzing only bounded or limited cases, thus reducing the strength of the obtained verification results). The former has been the main cause for delaying the effective use of automated formal methods directly on software (e.g., applying them to software designs rather than to actual code), and the latter is the reason why some automated formal methods are called *lightweight*, to contrast with the *heavyweight* ones relying mainly in formal deduction as a mechanism for analysis. The relevance of lightweight formal methods, and in particular of their direct application to code rather than designs and other more abstract representations, is now especially important, considering today's "agile" world with fewer designs and intermediate notations, and the further concentration of development activities in code. Various formal methods, and in particular lightweight ones, are then continuously trying to better tackle code.

In this talk we will discuss a particular *lightweight* formal automated technique, that we have been using for more than 15 years to directly analyze code. The so-called technology for analysis is SAT-solving, i.e., the process of automatically deciding a propositional formula's satisfiability. Besides discussing how various relevant problems in software engineering can be encoded as boolean satisfiability cases, we will describe how quickly these problems become too complex to be directly handled by SAT solvers, and our approaches to at least partially overcome the scalability and thoroughness issues mentioned before. We will discuss three main problems of increasing complexity in software engineering, namely test generation, bounded verification (the lightweight counterpart of the traditional verification problem), and a problem we have more recently started working on, automated program repair. Besides the particularities of these problems in our context, and the details of how they are handled, we will describe a number of techniques to increase scalability, that we believe can be generalized to be applied in other contexts. These range from techniques that deal with solvers in a black-box fashion, e.g., improving analysis by producing simpler formulas to be fed to solvers, to techniques that "tamper" directly with how SAT solvers proceed when checking formula satisfiability.

Finally, during the last part of our talk we will discuss again the move from heavyweight to lightweight formal methods, and the increasing use of what some researchers have called *invisible formal methods*, i.e., the use of formal methods in helping to solve problems of informal software development approaches. We will discuss some risks of taking the route toward lightweight formal methods too far, in particular eliminating the need for formal specification. We will argue that while formal specifications may be considered accessory in some contexts, they are essential for difficult analysis problems, in particular automated program repair.

This is joint work with Marcelo Frias and other contributors.