Formal methods and static analysis: an overview

David Monniaux
28 February 2012

Abstract

Formal verification of hardware and software was once considered an academic pursuit remote from practical industrial use. Yet, a number of verification systems have made inroads, particularly for safety-critical software. We shall give an overview of the major classes of techniques, their pros and cons, geared towards engineers interested in building safer systems at reasonable costs.

Software contains bugs; this is largely an accepted fact of life, and users put up with desktop applications crashing and cell phones freezing and rebooting. In the case of safety-critical software, as found in, for instance, aircraft, public transportation systems, or medical devices, such failures are intolerable.

Traditionally, bugs in critical software are addressed through a combination of procedural requirements on the development process (thus addressing only indirectly the quality of the final software) and intense testing. Such processes can yield high quality software with infrequent failures, as witnessed by the safety of modern aircraft systems, but at high cost: maintaining huge test bases requires considerable human manpower, and programming guidelines and safety considerations may impose costly design constraints, which themselves may have negative safety implications (e.g. if one does not trust optimizing compilers, one runs unoptimized object code, which is slower, thereby needs faster processors, posing electrical and thermal problems).

An alternative is to prove by mathematically sound methods that the software cannot produce certain kinds of undesirable behaviours. Proof techniques for programs have been studied since the seminal work of e.g. Robert Floyd and Tony Hoare in the late 1960s. Modern proof assistants include PVS\(^4\), Coq\(^5\), ACL\(^3\), and Isabelle\(^4\) but these tools operate over their own internal programming languages. The Frama-C\(^5\) and the KeY\(^6\) systems allow users to do assisted proofs on C and Java respectively. Assisted proof stills tends to be overly tedious and human-intensive; for instance, it generally requires that the user provides useful inductive invariants for all loops and recursive functions.

In contrast, static analysis tools tend to be fully automated. The vocabulary here is somehow misleading, since different groups define this phrase differently. To some, static analysis includes syntactic tools such as the traditional Unix lint command, which mostly flags stylistic errors and known hazardous constructs from the programming language.
(e.g. writing `while (condition); { ... }` is probably an error). Such tools may both flag issues that do not result in problems in reality, and omit real problems. Stronger tools in the same vein include Splint[1] and Coverity[2]. Such tools are well-tailored for finding bugs in corporate applications, where bugs tend to be numerous — thus such applications are generally tailored towards efficiency in finding commonplace bugs rather than proving their absence.

For safety-critical systems, another class of static analysis is needed, which only provides *sound* results, that is, cannot omit mentioning a possible error if this error truly can happen in the real system. Basically, such analysers compute a “bounding box” — in technical terms, an *inductive invariant* — that includes the initial states of the system and is stable by moving to the next program state, and thus includes all states reachable by the program. The tool then checks that these states do not include erroneous conditions. As a practical example, faced with an array access `t[i]=3;`, such a static analysis tool may compute an interval `[a, b]` for variable `i`, guaranteed to hold every time this line is executed, and checks that `a ≥ 0` and `b` is less than the length of array `t`. Examples of such tools are the PolySpace verifier[7] now developed and marketed by The MathWorks, Astére, developed by CNRS, École normale supérieure and INRIA, and now sold through Absint GmbH[9] and the SPARK Ada static analyser.

None of these sound tools are *complete*: that is, they may fail to prove that some safety property (e.g. “there is no array overflow at line 100”) holds even though it holds in reality, leading to “false alarms”: warnings about problems that cannot occur in the real system. In fact, it can be shown mathematically that it is impossible to have a static analysis method that is both sound and complete (that is, only proves true properties and never fails to prove a true property) and that does not put arbitrary restrictions (memory, running time) on the analysed program[9]. The reduction of the number of false alarms encountered in practice, as well as of the analysis costs (time, memory), are active research pursuits.

Finally, there exist *bounded model checking* tools that check whether some violation is reachable from program start in a given number `N` of steps. Barring special circumstances (such as the running time of the program being bounded by a low constant `N0`), Such a tool cannot prove the absence of violations: if no violations exist at depth `N`, it does not follow that there are none at depth `N’ > N`. Some variants of these tools recover soundness by applying *predicate abstraction*, as in Microsoft’s Static Device Driver Verifier[9] or `k`-induction, as in the tool Kind[9].

There is no “one size fits all” answer as to whether static analysis and other formal methods are suitable for software development. This depends on the kind of software (the more dynamic, the more versatile, the more difficult to analyse, in general), the kind of properties to check (absence of runtime errors vs correctness of the software with respect to a functional specification), and the cost of a bug (loss of life and limb vs some lost desktop application data).

---

[5] This result, given formally in the mathematical setting of computability theory, is known as Rice’s theorem and is proved by reduction to Turing’s halting problem.