Capabilities and Limitations of Static Error Detection in Software for Critical Systems

S. Tucker Taft,
SofCheck, Inc.
Burlington, MA

A new category of static analysis tool is beginning to appear in the marketplace that goes beyond checking of relatively local characteristics of software modules to assess the logical consistency of an entire software system. Our belief is that these tools have the potential to dramatically enhance the effectiveness and timeliness of validation and verification of software for critical systems. Nevertheless, these tools have limitations, and it is important to understand both the capabilities and limitations of such tools to ensure that they are utilized most productively in the overall development life cycle of software for critical systems.

Thanks in large part to dramatic increases in computational horsepower available today to software development organizations, it is now practical to do significant static checking of large software systems. Although the theories underpinning static checking of software go back several decades, the intense computational requirements of these theories when applied to large software systems has made them more of academic than industrial interest. However, today the hardware has caught up with the theory, and million line software systems are now amenable to sophisticated static analysis.

There are various techniques for advanced software static analysis, but they can be broken down into roughly three general approaches, one which is based on program proving theory, one which is based on model checking, and one which is based on flow analysis and abstract interpretation. Historically, program proof engines have required human interaction to avoid getting stuck when attempting to handle loops or recursion. This has made them less appropriate for the analysis of large systems, which might requires hours of computation where periodic human interaction would likely be impractical. Advances in program proving are beginning to create useful fully automated proof engines, and applying program proof techniques to large systems may become possible. Another limitation of program proving approaches can be the requirement for a precise specification of what the program is doing. This simply doesn't exist for many large systems, or it is expressed in a kind of structured English that may not be amenable to automated analysis.

The model checking approach originated in tools for validating hardware designs, and such tools can have scalability issues when applied to large software systems, because of their dependence on examining the entire state space. Even for hardware systems, tricks are often necessary to reduce the size of the state space. For software systems, transforming the program into a "boolean" program or other techniques to dramatically reduce the state space are generally the first step in any model checking process. Another issue with model checking is it tends to be limited to verifying a single predicate at a time, and if the predicate is not verifiable, it may only find a single error rather than all of the errors that cause the predicate to be false.

Flow analysis with abstract interpretation is a somewhat more flexible approach, and is similar to the kinds of analysis being performed on a routine basis by optimizing compilers. As such, it tends to be a better fit to the large state spaces of software, and incorporates relatively straightforward approximation techniques to deal with structures that could lead to very large numbers of iterations. The major difference between what typical optimizing compilers do and what advanced static analyzers do relates to the level of interprocedural analysis. Compilers tend to limit interprocedural analysis because the benefits in terms of object code performance are often outweighed by the costs of full optimization. For static analysis, on the other hand, object code performance is not the measure, but instead the goal is to find logical inconsistencies or suspicious code that might indicate a design or coding error. In addition, it is adequate to do the static analysis perhaps only once a day, while compiling is often being done hundreds of times a day, so the performance of the tool itself is not as critical. Hence, the cost-benefit ratio tends to favor a higher level of interprocedural analysis, ultimately achieving full program or even full distributed system analysis.

While program proving can potentially show conformance to precise specifications of functionality, the other two approaches tend to focus on validating more generic properties relating to logical consistency, and freedom from undesirable behaviors such as race conditions, buffer overflows, memory leaks, dangling references, uses of uninitialized memory, etc. In one sense, the big shift in thinking that has enabled more progress to be made in automated static error detection has been to bypass the need for precise specifications inherent in proving a program correct, and focus on properties that are desirable or undesirable independent of the particular application -- that is, focus on proving the program does not perform patently incorrect or meaningless actions. This allows the tools to perform checks that are based on the underlying semantic rules of the language or machine, without the application programmer having to provide a detailed specification. Even with the program proving approach, proofs may focus on proving freedom from bad actions rather than presence of correct functionality. And even with the other approaches, some degree of crossover from application independent analysis to application-specific analysis can be provided if there are programmer-provided assertions present in the code, as the model checker or flow analysis engine can attempt to prove the truth of such assertions.

Any of the approaches can suffer from either false “negatives” or false “positives.” By this we mean that the approach might miss certain errors, or might identify constructs that are in fact not errors. In general, these “imprecisions” in the analysis are due to the approximations being performed to enable the analysis to proceed. Although program proving tends to make the fewest approximations, there may still be times when an imprecise assumption needs to be inserted to allow the proof to complete. Model checking tends to require approximations as part of reducing the size of the state space to be explored. Flow analysis with abstract interpretation tends to require approximations to allow iterative algorithms to reach a fixed-point. Note that even when there are no false positives, there might be constructs that could fail eventually (such as a loop counter overflowing), but which will not fail in the typical lifetime of the system.

Generally, if the analysis is considered “sound,” then there should be no false negatives for the kinds of errors that are detected. However, there are cases where a “sound” analysis may produce an unacceptable level of false positives while an “unsound” analysis would produce a more acceptable number of both false negatives and false positives. A common approach is to rank the identified problems by likelihood of their being a “true” error, as opposed to a false positive. In this way, the user may choose to focus only on the messages with a very high likelihood of being real, while ignoring those where the likelihood is smaller, intentionally choosing to risk the possibility of a false negative due to not examining all low-likelihood messages.

Other sources of either false positives or false negatives can be due to capacity limitations, requiring partitioning of the system under analysis. Some of the techniques suffer from nonlinear time or space scaling as the system grows larger, and even with those that can come close to linear scaling, some practical limits in terms of time, memory space, or disk space may ultimately limit the size of system that can be analyzed when accounting for all interprocedural dependencies.

An important differentiator in analysis tools is whether they take a fundamentally “top-down” or “bottom-up” approach. With a top-down approach, the tools start at one or more entry points to the system, and then check whether code can be reached with values that might cause bad behavior to occur. With a bottom-up approach, the tools start at the “leaves” of the system, modules that don’t depend on any other module, and then work up toward the higher levels of the system, characterizing each module in terms of its pre- and postconditions, and checking those at the point of each call. The bottom-up approach tends to be more scalable, but can have more challenges in terms of false-positives, in that it doesn’t generally have information on the parameter values provided by callers before it attempts to determine whether a given called module is correct. And it is also possible to combine the two approaches in various ways, to perform multiple passes, iterate over the entire system, etc. One of the biggest advantages of a bottom-up approach is that it naturally adapts to analyzing pieces of a system, from a single module, to a subsystem, to a library of modules, up to an entire system. It also has the potential of providing useful as-built documentation in terms of pre- and postconditions derived from the actual code, rather than relying on programmers providing annotations or testing only against the current points of call, rather than against all future potential uses of the code.

An important practical consideration when choosing a static analysis tool is whether it can handle the source language(s) used in the system, including any idiosyncrasies of the language that might be peculiar to a particular target or compiler. For some languages, in particular Java and C#, there are relatively standardized representations, namely Java bytes codes and .NET IL, which can be used as a starting point for analysis. For other languages, such as C and C++, finding a tool that can handle the particular dialect in use on a given system may be a significant challenge.

Other practical considerations include what kind of explanation the tool provides when it identifies a problem, and how hard it is to determine whether a message identifies a problem that is of actual concern to the developers. Also, once a message has been determined to not be of interest, tools vary ini how difficult it is to prevent the message from reappearing in future analyses, either in exactly the same form, or in a similar form.

To conclude, our belief is that the new set of advanced static error detection tools deserve consideration in any software development organization that is building software for critical systems. However, the tools have limitations, and it is important to understand how those limitations might affect the value of the tool, and how well the tool matches or is adaptable to the particular needs of the development organization.