

Research Statement

Eric Bodden

Overview

The safety and security of large-scale programs has always been a concern for software developers, and has become one of the dominant topics in software development in recent years. My personal research goal is to provide easy-to-use and scalable tools that allow programmers and users to guarantee that programs fulfil crucial safety and security properties.

I approach this problem through a hybrid combination of static program analysis techniques and dynamic checkers, called runtime monitors. Given a safety or security property, I first apply precise static program analysis techniques to prove the program correct with respect to this property. If the static analysis fails to prove the program completely correct, then I use compilation techniques to modify the program under test so that the program will check itself for compliance with the property at runtime. The inserted monitoring code is guaranteed to capture any property violation just before it occurs, and can issue error messages, initiate a roll back or reset the program to a stable state. The monitoring code is also optimized. My approach makes sure to insert monitoring code only at the small fraction of program locations where the static analysis failed to prove the program correct. Secondly, the static analysis uses heuristic techniques from software engineering to report a list of potential property violations to the user, automatically ranked so that most likely violations appear first. Programmers can easily check this list to see if actual violations may occur, even without running the program under test.

Current Research: Hybrid static and dynamic program verification

Expressive multi-object properties

My current verification approach allows programmers to statically verify and dynamically check properties that involve one or multiple objects (similar to tpestate properties), and that can be expressed as a finite-state machine. A typical single-object property, could be that for every file f , if the program under test has protected f , this program accesses f only after having been granted permission to do so. A typical multi-object property could be that one does not modify a hash map h while at the same time iterating over h 's key set k with an iterator i .

I originally focused my analysis approach on a specification language called tracematches [1]. Tracematches allow programmers to define properties via regular expressions. A possible regular expression for the single-object property stated above could be “File f : `protect(f) access(f)`” over the alphabet $\{\text{access}(f), \text{grantAccess}(f), \text{protect}(f)\}$. This regular expression matches whenever an access to f occurs after f was protected but without access to f being granted in between.

Scalable staged static program analysis to identify possible property violations

My static analysis [5] is staged in order to reduce the necessary analysis time. Earlier stages execute very fast and can rule out many potential violations cheaply, while later stages are much more involved. Because the earlier stages usually perform the bulk of the work, the later stages execute relatively quickly as well, once they are executed.

The first stage investigates the program under test to see if this program can at all violate the stated property. In the example, if the program never protects or accesses any file at all, then surely it is correct with respect to the property. The second stage applies a pointer analysis, based on the following observation. The program is also correct, if it never accesses a protected file f , or if it never protects a file f that is being accessed. Even if the analysis cannot prove the program completely correct, then it can prove that no violations can take place for such files f . In a last step [6], the analysis inspects the possible orderings in which events on f can occur. For instance, the program is allowed to access f before it was protected and after access to f has been granted, but not in between.

Treating actual violations at runtime, and identifying likely violations statically

If the analysis determines that violations could potentially occur at runtime, it gives the programmer two options to conveniently handle these potential violations. Firstly, I use special compilation techniques to modify the program under test so that it notifies a runtime monitor. This monitor monitors the program's execution at runtime, but only at these program locations which the static analysis was unable to prove correct. The resulting instrumented program often executes several orders of magnitude faster than a fully instrumented program could execute. For difficult cases, where instrumentation has to be inserted at program locations that are very frequently executed, I showed [4] that it is possible to distribute the instrumentation over multiple users, or to enable the instrumentation only at certain times. Both techniques ensure a low runtime overhead and improved user experience.

Secondly, the static analysis uses heuristics from software engineering to determine potential violations that are particularly likely to manifest themselves at runtime. I showed [6] that these heuristics can be over 99% accurate for some large-scale benchmarks. For a programmer it is easy to identify actual errors by inspecting the few program points where violations are likely to occur.

Extension to finite-state specification formalisms in general

In recent work [3], I showed that the first two of my three static analysis stages can in fact be applied not only to properties stated in the form of tracematches, but they can also easily be applied to properties stated in any other specification formalism that can be reduced to a finite-state machine. In fact, several runtime monitoring tools exist which allow for the translation of property specifications written in such finite-state formalisms into aspect-oriented monitoring code. Programmers then weave this monitoring code into their program to enable the runtime monitoring of the stated properties in their programs. One of these code generation tools is JavaMOP [2], supporting specifications in past-time and future-time linear temporal logic and extended regular expressions. We modified JavaMOP such that it encodes domain knowledge, which is present in the finite-state specification, directly in the generated aspect code, in the form of annotations. My staged static analysis can then exploit this domain knowledge to conduct the first two analysis stages, and to optimize the program instrumentation. The approach yields the same significant speedups that we already observed [5] for tracematches earlier.

Future research

My hybrid analysis approach succeeds in enabling programmers to identify programming errors early in the development process, through static analysis, and in providing safety and security guarantees about programs at runtime, through runtime monitoring. Nevertheless, the approach still suffers from some practical limitations which I would like to address in the future.

Short-term goals: Multi-threading and generalization of flow-sensitive analysis

Virtually all tpestate-like analyses, including my own third, flow-sensitive analysis stage, assume a single-threaded program, because this eases the reasoning about control flow. However, many modern programs are multi-threaded and therefore invalidate the analysis results. Fortunately, my analysis is flow-sensitive on an intraprocedural level only [6]. This makes it comparatively easy to make my analysis thread-aware and sound also for multi-threaded programs. This could involve using may-happen-in-parallel analyses or thread-local objects analyses which have been proposed in the past.

Another short-term goal includes opening up this third, flow-sensitive analysis stage so that, it also works for finite-state properties in general, not only for specifications in the form of tracematches. This would likely require encoding dominance relationships between transitions in finite-state machines in the generated aspect code, similar to the approach that I took in [3]. The third analysis stage could then use its flow-sensitive information to validate or contradict these dominance relationships.

Long-term goals: Incremental and modular analysis, and property inference

My primary long-term goal for future research is to make the analysis approach modular and incremental. My current approach requires the whole program to be available and analyzed, which is often impractical

and time consuming. A key observation is that very often, safety properties are specific to a given application interface (API). For instance, the multi-object property on hash maps and their key sets mentioned above can be seen as part of the specification of the API of the Java Runtime Library. It would suffice to verify the Java Runtime Library itself for compliance with that property only once. My goal is to develop an approach to embed both properties and their compliance proofs directly in a library's interface. My analysis could then use the embedded information through assume/guarantee reasoning: assuming that the library code has already been proven correct, it can guarantee the correctness of client code easier and faster. In addition, because the library code is shipped directly with the safety properties that state how its API ought to be used, clients can verify their compliance with these API contracts completely automatically.

A secondary long-term goal addresses the problem of obtaining property specifications in the first place. My current approach assumes that specifications are given. Determining such specifications is a non-trivial task. Previously, I determined specifications by manually inspecting the source code, comments and documentation of well-known APIs. Instead, one could infer specifications in the form of API contracts directly from an API's implementation. For instance, one could infer preconditions that have to be met in order to prevent certain exceptions from being thrown. While initial research in this direction has already been conducted by others, the contracts that current approaches can infer are often simple and incomplete, or the inference involves manual steps. I plan to design and implement scalable and fully automated analysis techniques that are able to infer expressive multi-object properties.

References

- [1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM Press, October 2005.
- [2] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588, 2007.
- [3] [Eric Bodden](#), Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Proceedings of the 8th international conference on Aspect-oriented software development (AOSD)*, March 2009. To appear.
- [4] [Eric Bodden](#), Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. *Oxford Journal of Logics and Computation*, 2008. To appear.
- [5] [Eric Bodden](#), Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 525–549, 2007.
- [6] [Eric Bodden](#), Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)*, pages 36–47, New York, NY, USA, 2008. ACM.