Collaborative runtime verification with tracematches

Eric Bodden[1], Laurie Hendren[1], Ondřej Lhoták[2], Nomair A. Naeem[2]

[1] McGill University, Montréal, Québec, Canada
[2] University of Waterloo, Waterloo, Ontario, Canada

In the verification community it is now widely accepted that, in particular for large programs, verification is often incomplete and hence bugs still arise in deployed code on the machines of end users. Yet, in most cases, verification code is taken out prior to deployment due to large performance penalties induced by current runtime verification approaches. Consequently, if errors do arise in a production environment, bugs are hard to find, since the available debugging information is often very limited.

In previous work on *tracematches* [1], we have shown that in many cases runtime monitoring can be made much more efficient using static analysis of the specification [2] and program under test [3]. Most often, the imposed runtime overhead can be reduced to under 10%. However, the evaluation we performed also showed that some classes of specifications and programs exist for which those optimizations do not perform as well and hence large overheads remain. According to researchers in industry [5], larger industrial companies would likely be willing to accept runtime verification in deployed code if the overhead is below 5%. Hence, additional work is required in order to make runtime verification scale even better.

In this work, we tackle this problem by applying methods of remote sampling [4] to runtime verification. Remote sampling makes use of the fact that companies which produce large pieces of software (which are usually hard to analyze) often have access to a large user base. Hence, instead of generating a program that is instrumented with runtime verification checks at all necessary places, one can generate different kinds of partial instrumentation ("probes") for each such user. A centralized server then combines results of all runs of those users. This method is generally very flexible. In particular, we see the following advantages over a complete runtime verification.

Less runtime overhead per user. The program each user runs is only partially instrumented and hence the instrumentation overhead can be kept to a moderate level.

Better coverage of relevant paths. In order for runtime verification to be complete, perfect path coverage is necessary. In general, this is nearly impossible to achieve. If instrumentation could be dynamically adapted, it could be focused on paths that are actually being executed during users' program runs.

Assigning priorities. Similarly, usage data could be used to assign priorities to bugs that are triggered by many users.

Automatic analyses. The server that receives the event data in the end can apply arbitrarily sophisticated analyses on the received data and automatically attach this information to a bug report. This is in contrast to existing error reporting systems, which are mostly operated manually.

In this work we focus on the first part, reducing the runtime overhead, and present experiments for providing such an infrastructure based on static compilation of tracematches. Since tracematches allow for per-object specifications via free variables, special attention has to be paid to object bindings. Using a flow-insensitive whole-program analysis proposed in [3], we obtain groups of related instrumentation points which need to be triggered at runtime in order to obtain a property violation. Each probe is defined as such a set of instrumentation points. We extended our compiler such that each probe is guarded by a Boolean flag whose status can be dynamically changed.

As we will show, it is safe to freely enable and disable probes while still preserving the correct tracematch semantics. In general, this approach gives up completeness, though. Hence, we explain how techniques from Liblit et al. [4] can be used to express probabilities with which a given piece of software is correct if no errors are detected. In the situation where there exist at least as many users as different probes and if probes are evenly distributed amongst those users, this probability can amount up to 100%. In those cases no precision is lost with respect to a fully instrumented program.

In order to prove the feasibility of our approach, we applied our modified compiler to some of our largest benchmarks from previous evaluations [3]. Our results show that in many cases, the instrumentation overhead can indeed be lowered from as much as 250% to less than 10% (for each user). We also identified two possible sources of problems. As mentioned in [3], under some unfortunate circumstances (imprecise points-to sets, very long-lived objects) probes can become larger than usual. Consequently, in those cases the instrumentation overhead per user might be higher. Secondly, the reconfigurable instrumentation which we statically insert may impede other static optimizations due to the introduction of a more complicated branching structure. We discuss approaches to overcoming those problems as well as the possibility of dynamic reconfiguration of probes on the level of a Java virtual machine.

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 *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
2. Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondřej Lhoták, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, http://www.aspectbench.org/, 03 2006.
3. Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. Technical Report abc-2006-4, http://www.aspectbench.org/, 12 2006.
4. Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.
5. Wolfgang Grieskamp (Microsoft Research), January 2007. Personal communication.