The design and implementation of formal monitoring techniques

Eric Bodden

Sable Research Group, School of Computer Science, McGill University
eric.bodden@mail.mcgill.ca

Abstract. In runtime monitoring, a programmer specifies a piece of code to execute when a trace of events occurs during program execution. Previous and related work has shown that runtime monitoring techniques can be useful in order to validate or guarantee the safety and security of running programs. Yet, those techniques have not yet been able to make the transition to everyday use in regular software development processes. This is due to two reasons. Firstly, many of the existing runtime monitoring tools cause a significant runtime overhead, lengthening test runs unduly. This is particularly true for tools that allow reasoning about single objects, opposed to classes. Secondly, the kind of specifications that can be verified by such tools often follow a quite cumbersome notation. This leads to the fact that only verification experts, not programmers, can at all understand what a given specification means and in particular, whether it is correct. We propose a methodology to overcome both problems by providing a design and efficient implementation of expressive formal monitoring techniques with programmer-friendly notations.

1 Problem Description

Static program verification in the form of model checking and theorem proving has in the past been very successful, however mostly when being applied to small embedded systems. The intrinsic exponential complexity of the involved algorithms makes it hard to apply them to large-scale applications. Runtime monitoring or runtime verification [1] tries to find new ways to support automated verification for such applications. This is done by combining the power of declarative safety specifications with automated tools that allow to verify these properties not statically but dynamically when the program under test is executed. Researchers have produced a variety of such tools over the last years, many of which have helped to find real errors in large-scale applications.

Yet, those tools have not yet had any widespread adoption by programmers in real software development processes. In our opinion, this is mainly due to two reasons. Firstly, there is an obvious trade-off between expressiveness and complexity of any given runtime monitoring tool. The early tools were very lightweight, allowing users to specify properties such that *the method File.read(..) must be called only before File.close()*. Such properties can be checked very efficiently. However, in an object-oriented setting, usually only per-object specifications make sense: *for all Files f, f.read(..) must be called only before f.close().* Implementing runtime monitoring for such properties efficiently is a real challenge and until now the few tools which allow for such kind of specifications still induce an unduly large runtime overhead in many cases [4].

Secondly, many runtime verification tools build up on the mental legacy of static verification. In static verification, formalisms such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) are very common. Even experts in formal verification admit that those formalisms are often hard to handle.Hence, it is only natural that many programmers perceive runtime monitoring as complicated and consequently not very practical. In addition, even if one programmer decides to go through the process of learning such a specification language, he might not be able to communicate specifications he wrote to any of his colleagues, making potential mistakes harder to spot.

This lack of adoption of runtime monitoring techniques therefore leads to the fact that in most software development projects formal verification simply does not take place. Instead, hand-written tests are produced; a process which in itself is tedious and error-prone. As a consequence, the potential of those powerful techniques just remains unused, leaving many faults, which otherwise could have been detected, quietly buried in program code.

2 Goal Statement

Our goal is to evolve the techniques of runtime monitoring to such a state that they can *easily be used* by *reasonably skilled programmers* on *large-scale applications* written in *modern programming languages*. Specifically, we want to tackle the problems of (1) efficient runtime monitoring for parametrized specifications and (2) providing specification formalisms that can easily be understood by programmers and can be used to not only verify runtime behaviour but also communicate design decisions between developers.

To ease the software development process, our approach should be automated as much as possible. Therefore, such a tool chain would have to consist of the following components:

– A front-end that provides support for denoting safety properties in a variety of (potentially graphical) specification formalisms.
– A generic back-end that allows for the automatic generation of runtime monitors for any such formalism. The generated monitors should be as efficient as possible, even if specifications are to be evaluated on a per-object basis.
– A static analysis framework to specialize instrumentation code with respect to the program under test. Goal of this framework is to remove any instrumentation overhead induced by the monitor, in case this overhead can statically be proven unnecessary.

This tool chain would address the stated problems in the following ways. The potentially graphical front-end would allow programmers to denote safety properties in a way that is close to their mental picture of it. Bridging this gap between what the programmer wants to express and needs to express is essential in order to guarantee a minimal chance for error at specification time. The front-end should potentially be integrated into an existing integrated development environment in order to provide programmers easy access.

The back-end then generates efficient code from those safety properties. At this stage, high-level events are mapped onto events in the actual code base. From our experience we can tell that one potentially good way of doing this would be to use *pointcuts* of an aspect-oriented programming language [6]. Such pointcuts have proven themselves to be easy enough to understand for many average software developers, as their wide-spread use in software development proves.

Although the back-end generates efficient code, this code might still not be efficient enough, in particular in scenarios where the program under test would trigger the generated runtime monitor very frequently. Static program analysis can decrease the runtime overhead by statically determining that certain static instrumentation points can never be part of a dynamic trace that would trigger a violation of the given specification. The static analysis back-end should be able to offer generic analyses that can be applied to specifications in arbitrary formalisms and flexible enough to allow additional formalism-specific analysis stages to be plugged-in. Furthermore it must be capable of automatically specializing the generated monitor based on the gathered analysis results. In order to make sure the overall goal is reached, we propose the following methodology.

3 Approach

Efficient monitor code generation: We base our approach on an already developed back-end for *tracematches* [2]. Tracematches are an extension to the aspect-oriented programming language AspectJ [3] which allows programmers to specify traces via regular expressions with free variables. Avgustinov et al. already identified and solved many of the problems of generating efficient monitor code [4], yet for some benchmarks large overheads remain.

Removal of unnecessary instrumentation through static analysis: In a second step (ongoing), we then design and implement a set of static analyses which allows us to remove unnecessary instrumentation induced by the presence of tracematches. Initial results seem promising, lowering the runtime overhead to under 10% in most cases [5].

Making code generation and analysis generic: In a third step we then plan to conduct a study that investigates how much both, those static analyses and the mechanics for efficient monitor code generation can be generalized. In particular, one has to answer the question of *exactly what information needs to be known about a given specification formalism or the given specification itself in order to make code generation and analysis feasible.* Once this study has been conducted, the implementation of both, code generation and static analysis, will be generalized accordingly.

Easier specification formalisms: In a fourth and final step we will then concentrate on specification languages. We plan to study existing, potentially graphical, notations in detail. For each such notation we wish to answer the question of *why it is easier or harder to understand than others.* Some particular formalisms could also only be suited for special kinds of specifications. In that case we wish to determine the essence of such specifications and *why they are harder to express in other formalisms.* Finally, we plan to provide a prototypical front-end that allows users to denote specifications in different formalisms which seem particularly suited for large-scale mainstream software development. The addition of another domain-specific language could help answer the question of *whether or not domain-specificity in this setting can make sense.*

3.1 Evaluation

Well-known benchmarks exist for the evaluation of runtime overheads and the precision of static analysis. In our work to date we made use of the DaCapo benchmark suite which consists of ten medium-sized to large-scale Java applications. We plan to conduct consistent experiments with this and other benchmark suites throughout the entire project. One major contribution of our work will be to provide a set of specifications that apply to those benchmarks along with a detailed account of how the various optimizations behave on those specifications and which of the specifications are actually violated by the programs.

With respect to specification formalisms, the question of how those could be evaluated best, remains still unclear at the current time. Even if one had access to subjects willing to try various formalisms and compare them on a subjective basis, it would be hard to guarantee internal and external validity due to potentially different background knowledge of the subjects and due to the large variety of formalisms to choose from. In general, we tend to believe that graphical notations could improve comprehensibility a lot. Yet, this might be hard to prove. Hence, we would be very grateful for comments on that matter.

References

1. *1st to 7th Workshop on Runtime Verification (RV'01 - RV'07)*, 2007. http://www.runtime-verification.org/.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
3. AspectJ Eclipse Home. The AspectJ home page. http://eclipse.org/aspectj/, 2003.
4. P. Avgustinov, J. Tibble, E. Bodden, O. Lhoták, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, http://www.aspectbench.org/, 03 2006.
5. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*. Springer, July 2007. To appear in *Lecture Notes of Computer Science*.
6. G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996.