Towards Reliable Concurrent Software

Marieke Huisman
University of Twente
m.huisman@utwente.nl

Abstract

As the use of concurrent software is increasing, we urgently need techniques to establish the correctness of such applications. Over the last years, significant progress has been made in the area of software verification, making verification techniques usable for realistic applications. However, much of this work concentrates on sequential software, and a next step is necessary to apply these results also on realistic concurrent software. In this abstract, we argue that current techniques for verification of concurrent software need to be further developed in multiple directions: extending the class of properties that can be established, improving the level of automation that is available for this kind of verification, and enlarging the class of programs that can be verified.

1 Introduction

Software is everywhere! Every day we use and rely upon enormous amounts of software, It has become impossible to imagine what life would be like without software. This creates the risk that one day software failures will bring our everyday life to a grinding halt. In fact, all software contains errors that cause it to behave in unintended ways [4, 6], and substantial research is needed to to help software developers to make software that is reliable under all circumstances, without compromising its performance.

A commonly used approach to improve software performance is the use of *concurrency* and *distribution*. For many applications, a smart split into parallel computations can lead to a significant increase in performance. Unfortunately, parallel computations make it more difficult to guarantee *reliability* of the software. The consequence is unsettling: the use of concurrent and distributed software is widespread, because it provides efficiency and robustness, but the unpredictability of its behaviour makes that errors can occur at unexpected, seemingly random moments.

The quest for reliable software builds on a long history, and significant progress has already been made. Nevertheless, ensuring reliability of efficient software remains an open challenge. Ultimately, it is our dream that program verification techniques are built into software development environments. When a software developer writes a program, he explicitly writes down the crucial desired properties about the program, as well as the assumptions under which the different program components may be executed. Continuously, an automatic check is applied to decide whether the desired properties are indeed established, and whether the assumptions are respected. If this is not the case, this is shown to the developer – with useful feedback on why the program does not behave as intended.

FM@Lorentz, 2018
2018.

2 Abstraction Techniques for Functional Verification

One of the main challenges for the verification of concurrent software that we see is to automatically verify *global functional* correctness properties of concurrent software. To reach this goal, we advocate an approach where a *mathematical model* of a concurrent application is constructed, which provides an *abstract view* of the program's behaviour, leaving out details that are irrelevant for the properties being checked [3, 7], see Figure 1. The main verification steps in this approach are

1. *algorithmic verification* over the mathematical model to reason about global program behaviour, and
2. *program logics* to verify the formal connection between the software and its mathematical model.

Typically, the basic building blocks of the abstract mathematical model are *actions*, for which we can prove a correspondence between abstract actions and concrete code fragments. A *software designer* specifies the desired *global properties* for a given application in terms of abstract actions. The *software developer* then specifies how these *abstract actions map to concrete program state*: in which states is the action allowed, and what will be its effect on the program state. Global properties may be safety properties, *e.g.*, an invariant relation between the values of variables in different components, or a complicated protocol specifying correct interface usage, but we believe that extensions of the approach to liveness and progress properties are also possible.

To further develop this approach and make it scale, we believe the following challenges should be addressed:

1. identify a good abstraction theory,
2. extend the abstraction theory to reason about progress and liveness properties of code, and
3. use the abstraction theory to guide the programmer to develop working code through refinement.

3 Automating the Verification Process

Another major challenge is how to automate the verification process. At the moment, program verification requires many user annotations, explicitly describing properties which are often obvious to developers. We believe that many of the

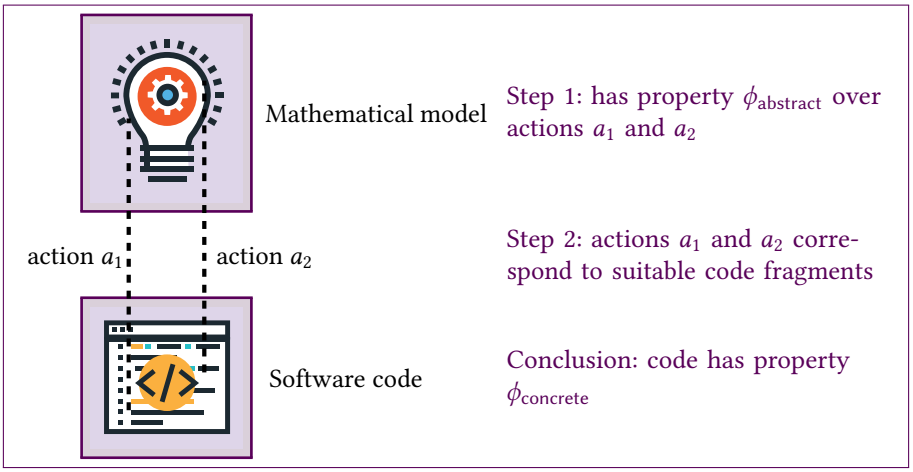

Figure 1. Using abstraction for the verification of concurrent and distributed software

required annotations can be generated automatically, using a combination of appropriate static analyses and smart heuristics.

We advocate a very pragmatic approach to annotation generation, where any technique that can be used to reduce the annotation burden is applied, combined with a smart algorithm to evaluate the usability of a generated annotation, removing any annotations that do not help automation. This will lead to a framework where for a large subset of non-trivial programs, we can automatically verify many common safety properties (absence of null-pointer dereferencing, absence of array out of bounds indexing, absence of data races etc.), and if we wish to verify more advanced functional properties, the developer might have to provide a few crucial annotations, but does not have to spell out in detail what happens at every point in the program (in contrast to current program verification practice).

4 Verification of Programs using Different Concurrency Paradigms

Finally, verification techniques need to support different programming languages, and different concurrency paradigms. In particular, we believe that it is important to investigate how to reason about programs written using the structured parallel programming model where all threads execute the same instructions. Recently, we have shown how our verification techniques can be adapted in a straightforward manner to GPUs (including atomic update instructions) [1, 2]. It turns out that the restricted setting of a GPU has a positive impact on verification: the same verification techniques can be used, and verification actually gets simpler. We believe that this direction should be explored further, as typical GPU programs are usually quite low-level, which makes them more error-prone.

An interesting extension of this work is to automatically transform a verified sequential program with annotations into an annotated GPU program, which will be directly verifiable [5]. We believe this idea can also be used for other compiler optimisations, such that they do transform not only the program, but also the correctness annotations, such that the result is a (hopefully) verifiable program again. Instead of proving correctness of the transformation, both the program and the annotations are transformed, such that after the transformation the resulting program with annotations can be reverified.

References

[1] S. Blom, S. Darabi, and M. Huisman. 2015. Verification of loop parallelisations. In *FASE (LNCS)*, Vol. 9033. Springer, 202–217.

[2] S. Blom, M. Huisman, and M. Mihelčić. 2014. Specification and Verification of GPGPU programs. *Science of Computer Programming* 95 (2014), 376–388. Issue 3.

[3] S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. 2015. History-based verification of functional behaviour of concurrent programs. In *SEFM (LNCS)*, Vol. 9276. Springer, 84 – 98.

[4] Archana Ganapathi and David A. Patterson. 2005. Crash Data Collection: A Windows Case Study.. In *Dependable Systems and Networks (DSN)* (2005-08-01). IEEE Computer Society, 280–285.

[5] M. Huisman, S. Blom, S. Darabi, and M. Safari. 2018. Program Correctness by Transformation. In *8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) (LNCS)*. Springer. To appear.

[6] Rivalino Matias, Marcela Prince, Lúcio Borges, Claudio Sousa, and Luan Henrique. 2014. An Empirical Exploratory Study on Operating System Reliability. In *29th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 1523–1528. https://doi.org/10.1145/2554850.2555021

[7] W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. 2017. An Abstraction Technique for Describing Concurrent Program Behaviour. In *VSTTE (LNCS)*, Vol. 10712. Springer, 191 – 209.