Speeding up the Proof Strategy in Formal Software Verification

Markus Wagner
School of Computer Science
University of Adelaide
Adelaide, Australia
markus.wagner@adelaide.edu.au

ABSTRACT

The functional correctness of safety- and security-critical software is of utmost importance. Nowadays, this can be achieved through computer assisted verification.

While formal verification itself typically poses a steep learning-curve for anyone who wants to apply it, its applicability is further hindered by its (typically) low runtime performance.

With the increasing popularity of algorithm parameter tuning and genetic improvement, we see a great opportunity for assisting verification engineers in their daily tasks.

CCS Concepts

•**Computing methodologies** → **Heuristic function construction; Randomized search;**

Keywords

Formal software verification, search-based software engineering, runtime improvement

1. INTRODUCTION

Formal verification is the act of proving or disproving that an algorithm or its implementation is correct with respect to its formal specification. The formal mathematical approaches include, among others, model checking, deductive verification, and program derivation [1, 3, 6].

While formal methods get increasingly used in industry in the development of safety- and security-critical programs, the general popularity of formal methods is not as high as it could be. Reasons include, among others, the steep learning curve (even for verification engineers), and the often unsatisfactory runtime performance.

Our goal is to increase the runtime performance of verification systems by improving their proof procedure. We see at least the following two different approaches:

1. automated algorithm parameter tuning to tune "magic numbers" used in existing proof procedures,
2. genetic improvement of the actual proof procedure, or of components thereof.

These two can be seen as extremes, with deep parameter tuning [10] sitting between.

Two success stories about the improvement of non-functional properties in the greater context of formal verification are the following. For example, Hutter et al. [7] reduced the runtime of the SAT solver SPEAR through algorithm parameter tuning. A different approach to the optimisation of a non-functional property was chosen by Bruce et al. [5], who modify the source code of the SAT solver miniSAT in order to reduce its energy consumption.

Similar success stories are rarely heard of, and we intend to change this by improving an open-source tool that is used in industry.

2. TARGET OF OPTIMIZATION: PROGRAM VERIFICATION SYSTEMS

Every program verification system has to perform (at least) two rather separate tasks: (a) handling the program-language-specific and specification-language-specific constructs, and reducing or transforming them to classical logic expressions, (b) theory reasoning and reasoning in classical logics, for handling the resulting expressions and statements over data types. One can either handle these tasks in one monolithic logic/system, or one can use a combination of subsystems. In our present research, we propose to focus on the second task.

To demonstrate the applicability of search-based software engineering for formal verification speed-up, we have chosen the KeY tool [1], an open-source verification system for sequential Java Card programs. In KeY, the Java Modeling Language (JML) is used to specify properties about Java programs with the common specification constructs like pre- and postconditions for methods and object invariants. Like in other deductive verification tools, the verification task is modularised by proving one Java method at a time.

In the following, we will briefly describe the workflow of the KeY system—in our case, we assume the user has chosen one method to be verified against a single pre-/postcondition pair. First, the relevant parts of the Java program (together with its JML annotations) are translated to a sequent in Java Dynamic Logic (JavaDL), a multimodal predicate logic [1]. Validity of this sequent implies that the program is correct with respect to its specification. Proving the validity is done using *automatic proof strategies* within KeY, which apply sequent calculus rules.

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908961.2931690

Results of a verification attempt in KeY are the following: either the generated JavaDL formula is valid and KeY is able to prove it; or the generated formula is not valid and the proof cannot be closed; or KeY runs out of resources.

By the way, this is not the first time that the KeY tool is subject to search-based software engineering approaches. For example, Beckert et al. [2] presented the first maximisation of a problem-specific test coverage criterion. Wagner [9] investigated this further and also minimised the time needed for regression testing by greedily choosing tests from a large set of automatically created tests. Recently, Bokhari et al. [4] further improved the test coverage results through problem-specific local search.

3. RUNTIME IMPROVEMENT OF KEY

As already mentioned, we are targeting the reasoning task of a verification system, which in KeY's case is handled by its automated proof strategy. This strategy decides which of the hundreds of rules are to be applied to an expression in order to efficiently show its validity. The difficulty here is that this strategy has grown over many years, with the input of many researchers from several universities. Currently, the strategy selection encompasses hundreds of lines of code, complex formulae, and about 350 numeric values to determine the next rule application.

In a first step to understand this historically grown selection procedure, we analysed the GIT history of the JavaDL strategy implementation. We focussed on changes in the cost function, when the comment indicated the change was due to some optimisation. This way, we determined ranges of the constants ("magic numbers"). As an example, we show 10 constants, their default value in the current KeY implementation, as well as in earlier versions, in order to provide some first idea about the ranges and the relative ordering:

parameter name	current default	previous defaults
pull_out_select	-1900	-2000
cut_direct	100	200, 10
gamma	10	
inEqSimp_expand	-4400	-4500
inEqSimp_directInEquations	-2900	-3000
inEqSimp_propagation	-2400	-2500
inEqSimp_pullOutGcd	-2150	-2250
inEqSimp_saturate	-1900	-2000
inEqSimp_forNormalisation	-1100	-1000
defOps_expandRanges	-8000	-5000

These 10 are currently considered by KeY developers to be the most influential in the proof procedure, however, we expect that others will prove to be important as well.

In the strategy selection approach, other costs are factored in, and in the end the strategy with the lowest cost (which can be negative) is applied.

Our next steps...

As the first step, we are preparing for KeY's strategy parameters to be optimised using sequential Model-based Algorithm Configuration (SMAC) [8]. Currently, KeY's test suite contains about 560 cases which run in approximately half an hour on a modern laptop. Of course, the underlying hypothesis is that the these tests (respectively their translations to proof obligations) exhibit some similarities that can be exploited in the parameter tuning. While this kind of algorithm tuning has been successfully demonstrated in other problem domains, we are not aware of its application to verification systems.

Another approach to speed-up the system would be to set up KeY in such a way that it generates proof obligations for other solvers such as the SMT solver Z3, of which the algorithm parameters could then be optimised as well. However, this is currently not possible due to technical issues.

Our long-term goal is to replace the complex and historically grown formulae in KeY's strategy selection with either an updated one (via genetic improvement) or with a completely new (evolved) one. Ideally, this process will become automated so that it can reoptimise the strategy calculations whenever features are added to KeY or when new regression tests are added. Ultimately, we plan to take the developers' minds off the strategy part so that they can focus on features, while verification engineers will enjoy an increase in runtime performance.

4. ACKNOWLEDGMENTS

We would like to thank our colleagues Thorsten Bormer and Mattias Ulbrich from the Karlsruhe Institute of Technology for providing the first insights.

This work has been supported by the ARC Discovery Early Career Researcher Award DE160100850.

References

[1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[2] B. Beckert, T. Bormer, and M. Wagner. Heuristically creating test cases for program verification systems. In *Metaheuristics International Conference (MIC)*, 2013.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer, 2010.

[4] M. A. Bokhari, T. Bormer, and M. Wagner. *7th International Symposium on Search-Based Software Engineering (SSBSE)*, chapter An Improved Beam-Search for the Test Case Generation for Formal Verification Systems, pages 77–92. Springer, 2015.

[5] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Genetic and Evolutionary Computation (GECCO)*, pages 1327–1334. ACM, 2015.

[6] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[7] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 27–34, 2007.

[8] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization (LION)*, pages 507–523, 2011.

[9] M. Wagner. Maximising axiomatization coverage and minimizing regression testing time. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2885–2892, 2014.

[10] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1375–1382. ACM, 2015.