# GUIDO: Automated Guidance for the Configuration of Deductive Program Verifiers

Alexander Knüppel
*TU Braunschweig, Germany*
a.knueppel@tu-bs.de

Thomas Thüm
*University of Ulm, Germany*
thomas.thuem@uni-ulm.de

Ina Schaefer
*TU Braunschweig, Germany*
i.schaefer@tu-bs.de

***Abstract*—The software industry is still in its infancy to widely adopt program verification tools as part of their daily software engineering processes. One key challenge is that many of today's program verifiers intent to cover numerous bug classes and are therefore *manually configurable* to support users with their varying verification projects. However, configuring a program verifier for a given verification problem requires extensive expertise, as an ill-chosen configuration may either unnecessarily slow down the verification process or even hinder a successful verification at all. In particular for configurable deductive program verifiers, this problem is barely addressed by current research. We propose GUIDO, a framework incorporating statistical hypothesis testing to compute promising configurations automatically. With GUIDO, domain experts channel their knowledge by formalizing hypotheses about the impact of choosing configuration options and let normal developers benefit.**

***Index Terms*—Formal verification, parameterization, recommendation system, formal methods, deductive verification**

## I. Introduction

A plethora of advances in the last decades of program verification techniques, such as model checking [1], deductive verification [2], or abstract interpretation [3], deliver reasons to believe that formal methods will play a vital part in future software engineering practices. However, as modern software systems dramatically increase in complexity and scale, a successful adoption by industry is still in its infancy. The shift from *theoretical research* to real application by *practitioners* is impeded by obstacles [4], [5], such as scalability issues, lack of training, and the need for explicit domain knowledge only available from a few experts. In order to address the latter obstacle, there is a firm belief that tool support for formal methods has to increase automation and decrease interaction [4], [6].

However, a challenge is that many automated program verification tools and frameworks are *configurable*. While there exist numerous techniques and heuristics for popular configurable model checkers (e.g., CBMC [7], CPACHECKER [8], VERIABS [9], and PESCO [10]), automatic configuration has seldom been considered in the context of deductive verification. Program verifiers based on theorem proving, such as KEY [2] or FRAMA-C [11], require user interaction to adjust their automated proof search, when the default configuration is insufficient. Accordingly, finding a suitable configuration for a given verification problem is difficult as expert knowledge in complex verification technologies (e.g., proof theory) is often necessary [12] to understand the influence of configuration options. Moreover, default configurations are often insufficient to verify software automatically [13].

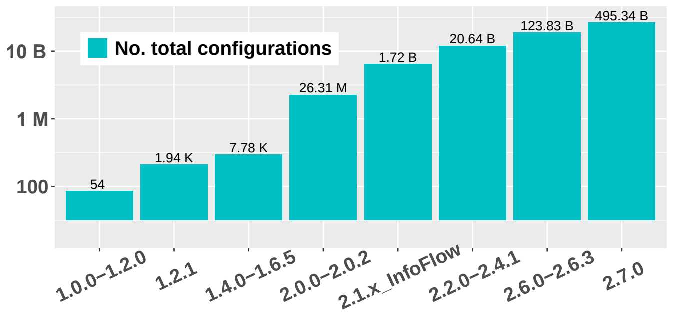


Fig. 1: Logarithmic scale of the total number of dissimilar configurations for all versions up to v2.7.0 of the deductive program verifier KEY [2].

Verification systems evolve over time, and the number of configuration options often increases with a new release. Consequently, users are faced with a combinatorial explosion in the number of possible configurations. For instance, the number of configuration options for the deductive program verifier KEY [2] increased from eleven (version 1.0.0) to 55 (version v2.7.0). As illustrated in Fig. 1, KEY v2.7.0 comprises about *half a trillion* different configurations. Practitioners may struggle to understand all the effects and influences that certain configuration options have on verification effort and success, and trying out all of them does not scale. Hence, in this work, we address the research question: *How can we make domain knowledge accessible in such a way that configuring formal verification tools becomes tractable even for less experienced users?*

For program verification, *promising* configurations can be described in two dimensions. The first dimension, *verifiability*, denotes whether a configuration is sufficient to verify a program against a specification (e.g., a method contract when following the *design-by-contract* paradigm [14]) automatically. Often, a verification procedure fails even though the input program is automatically verifiable, resulting in *spurious* failures that are hard to interpret or even to identify as such. Consequently, developers tend to refactor their programs or specifications instead of trying another configuration [15]. The second dimension, *verification effort*, is the effort needed to perform the verification task and may be measured differently depending on the optimization objective (e.g., execution time, proof size, or memory consumption).

While computing promising configurations looks like a natural optimization problem, it is unfortunately hard to address in practice. For instance, there exist numerous tools for performance prediction of configurable software based on regression techniques (e.g., SPLCONQUEROR [16], DEEPPERF [17], or DECART [18]). However, these tools are typically highly configurable themselves and primarily focus on performance for general-purpose applications, which poses a challenge for incorporating *domain knowledge* into the verification process. Consequently, these tools are inflexible for the special task of formal verification, where, in addition to verification effort, verifiability is a major concern.

**Contributions.** In this work, we thrive for practicality and propose the idea of a *two-phased* statistical framework for configurable deductive verification tools called GUIDO to find promising configurations systematically. In an *offline* phase, GUIDO takes as input a set of *hypotheses* (i.e., structured domain knowledge) defined by experts for their verification tool suite. Additionally, GUIDO needs access to a *benchmark data set*, which consists of a number of verification attempts for a set of configurations. Elements of that set include verification results, performance, and additional static analysis results of the input program (e.g., used language constructs and control flow properties). GUIDO then performs statistical tests on the benchmark data set with respect to the hypotheses to compute the costs of individual configuration options. In an *online* phase, GUIDO analyzes the input program and formulates a *constrained optimization problem* based on the analysis results and accepted hypotheses to compute an optimal configuration for the input program with respect to estimated costs. Hence, GUIDO aims at bridging the gap between two user groups: domain experts are formalizing hypotheses for their configurable verification tool individually, but only once (*offline phase*), and non-experts (e.g., software developers) benefit from the formalized domain knowledge and can apply GUIDO for their verification tasks automatically (*online phase*).

To get preliminary insights whether GUIDO helps to increase automation for configurable program verifiers, we have implemented a prototype and applied it to KEY [2] in version v2.7.0, a configurable and state-of-the-art deductive verifier for Java programs. KEY verifies Java programs specified with the *Java Modeling Language* (JML) [19], a behavioral specification language [20] following the *design-by-contract* principle [14].

**State of the Art.** The intent of GUIDO is to reimagine the challenge of parameter adjustment for software systems tailored to the context of formal verification. However, performance prediction of configurable software is a highly researched area. Siegmund et al. [16] proposed SPLCONQUEROR, a state-of-the-art framework using machine learning to measure and predict the performance of configurations. Despite its name, SPLCONQUEROR is used beyond *software product lines* by a multitude of researchers to estimate the influence of non-functional properties in configurable software [21]–[24]. Other performance prediction frameworks include CART/DECART [18], FOURIERLEARNING [25], or the recently published algorithm DEEPPERF [17]. While our approach requires the manual formulation of hypotheses, automatically applied general-purpose prediction frameworks fall short in at least three categories. First, they need more data, as they have to *learn* such hypotheses on their own. Second, they provide no simple explanation why a particular configuration option is more significant. Third, in case of a failed verification attempt, there is no continuation mechanism applied.

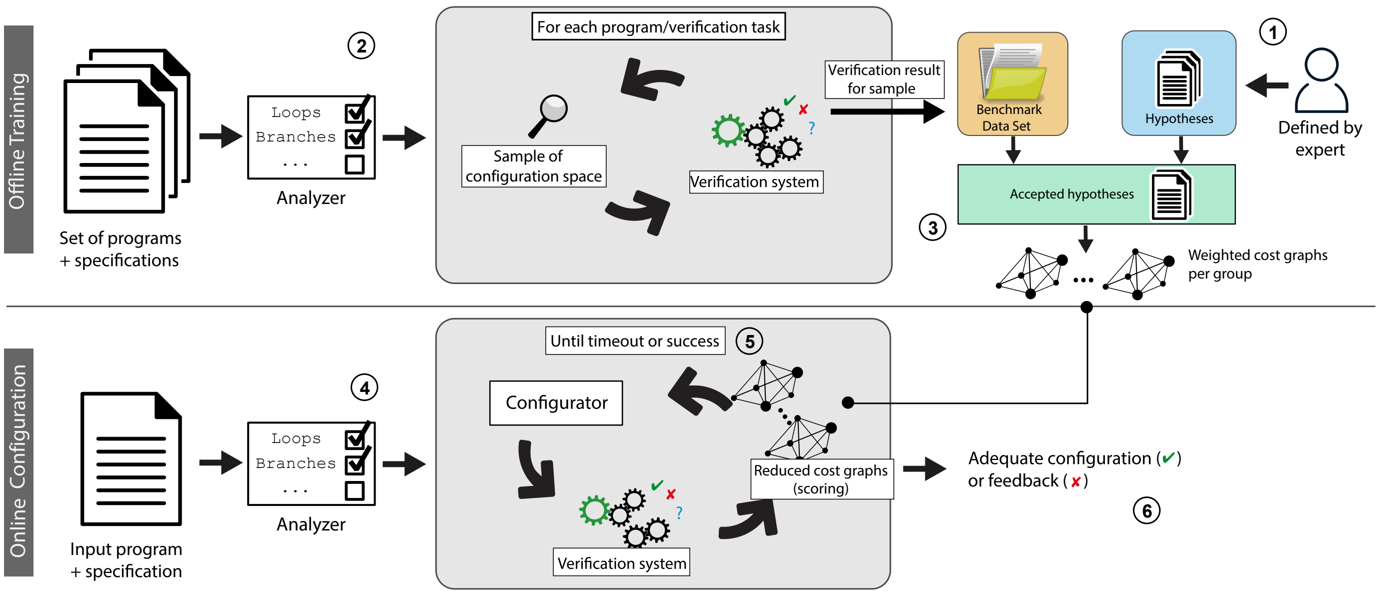
Regarding applicability of GUIDO, a plethora of configurable verification systems exists. For deductive verification, KEY is a program verifier for Java programs and was applied successfully to reveal serious defects in real production code [26]–[29]. Other configurable deductive program verifiers include FRAMA-C [11] and SPEC# [6]. The long-term goal of these tools aligns with GUIDO, which is to decrease interaction and increase automation in verification. In a previous study [13], we even hypothesized that there is a trade-off between verifiability and verification effort for KEY, which we aim to investigate in the future with GUIDO.

Another formal verification discipline controlled by a multitude of configuration options is model checking. Prominent and configurable model checkers include SPIN [30], Java Pathfinder [31], and CPACHECKER [8]. Our typical experience with model checkers is that most of the time experts are consulted for applying the optimal configuration, whereas our goal with GUIDO is to make verification tools more applicable to practitioners by providing them with a formalized version of such expert knowledge. In particular for model checkers, there exist numerous baselines (e.g., heuristics and default configurations) that can be used for a comparison with GUIDO.

## II. WORKFLOW OF GUIDO

Our main research goal is to build a practical framework that automatically predicts a promising configuration for configurable deductive verification tools. State-of-the-art tools for performance prediction based on regression analysis are not tailored for computing configurations that produce low verification effort while also guaranteeing high verifiability. First, the verification effort (i.e., performance) is not normally distributed [13], which is often a basic assumption for such tools. Second, both verification effort and verifiability are difficult to estimate in a black-box manner, as they typically depend on the structure of the specification and implementation to be verified. Third, collecting enough data points is difficult, as the adoption rate for verification tools by industry is rather low [4].

**Hypotheses.** To mitigate these issues, we propose to analyze the verification tasks in a white-box manner and to incorporate *domain knowledge* in the form of *statistical hypotheses*. The formalization of hypotheses helps us in two ways. First, instead of letting an algorithm completely learn on its own, we already encode partial knowledge about the effects of configuration options, which requires less data points, and should be more precise. Second, a formal foundation allows us to *generalize* our method to more configurable verification tools. The following example illustrates informally, how hypotheses about options may be expressed.



Offline Training
Set of programs + specifications
Loops
Branches
...
Analyzer
2
For each program/verification task
Sample of configuration space
Verification system
Verification result for sample
Benchmark Data Set
Hypotheses
1
Defined by expert
Accepted hypotheses
3
Weighted cost graphs per group
Online Configuration
Input program + specification
4
Until timeout or success
5
Configurator
Reduced cost graphs (scoring)
Adequate configuration (✓) or feedback (✗)
6

Fig. 2: Schematic workflow of GUIDO's offline- and online-phase to compute promising configurations automatically.

**Example: Hypotheses for KEY**

*Let $p_{\text{oss}} = \{o_{\text{d}}^{\text{oss}}, o_{\text{e}}^{\text{oss}}\}$ be a parameter labeled* `One Step Simplification`*, which is either* `Disabled` *($o_{\text{d}}^{\text{oss}}$) or* `Enabled` *($o_{\text{e}}^{\text{oss}}$). This parameter may enable the prover to reduce multiple proof steps into a single one during verification. Both options are* mutually exclusive*, such that exactly one of them has to be selected. Hypotheses focusing either on verifiability or verification effort may be formulated as follows.*

***Verifiability:*** *If a specification case is verifiable with option $o_{\text{e}}^{\text{oss}}$, it is also verifiable with option $o_{\text{d}}^{\text{oss}}$ (i.e., $o_{\text{d}}^{\text{oss}}$ is at least as effective).*

***Verification effort:*** *If the code to be verified contains loops, the verification effort with option $o_{\text{d}}^{\text{oss}}$ is at least as large as with option $o_{\text{e}}^{\text{oss}}$ (i.e., $o_{\text{e}}^{\text{oss}}$ is at least as efficient for loop-containing programs).*

*Whereas the first hypothesis on verifiability is generally applicable, the second hypothesis on verification effort only applies in the presence of loops.*

Ideally, we want to identify configurations that provide a high degree of verifiability, but also result in low verification effort. However, there exist reasons to believe that both criteria are on opposite sites of a continuum [13] (i.e., a configuration option may either improve the verification effort or verifyability, but typically not both). Our practical idea is to purposefully formulate hypotheses that support the search for configurations with a reasonable trade-off.

**Online and Offline Phases.** Fig. 2 presents an overview of GUIDO, which is divided into an *offline* training phase and an *online* configuration search phase. The offline training phase mainly consists of three steps and has to be performed for each major release of a verification system only once. First, a set of hypotheses on how specific configuration options influence the verification result regarding verifiability and verification effort has to be formalized (1). This task is best performed by domain experts, but can also be accomplished by analyzing and interpreting tool tips, documentation, or publications [13]. Second, a benchmark data set is generated by applying varying configurations for the verification of programs with varying language and specification constructs (2). As the configuration space may become too large for highly-configurable verification systems, GUIDO *samples* over the configuration space for the verification benchmarks. For industrial contexts, we assume that already enough verification projects and benchmark data are available to be used as input data for GUIDO. Third, all formulated hypotheses are tested using the benchmark data set to identify the set of accepted hypotheses given a corrected significance level. For this, we either apply a *McNemar test* [32] for hypotheses regarding verifiability or a non-parametric Wilcoxon test [33] for hypotheses regarding verification effort. Weighted cost graphs are computed that capture the influence of the configuration options on verification effort and verifiability as formulated by the accepted hypotheses (3).

In the *online* configuration search phase, GUIDO predicts promising configurations for new verification tasks. A user provides a specified program as input and, similar to the first phase, GUIDO extracts static analysis results, such as language and specification constructs (4). GUIDO then uses the created cost graphs from the offline phase and applies a score function to rate each configuration option individually with respect to the input program's characteristics. The total score function $\theta : o \to \mathbb{R}$ ranks an option $o$ higher compared to its alternatives if there is evidence given by the accepted hypotheses that $o$ performs significantly better. To compute the total score function $\theta$, we define two score functions, one function $\theta_{\text{ver}}$

TABLE I: Statistics on the Application to KeY v2.7.0

| Statistic | Value | Statistic | Value |
|---|---|---|---|
| Configuration options | 87 | Fixed options | 15 |
| Parameters | 30 | Significance level $\alpha$ | $\frac{0.05}{72}$ |
| Defined hypotheses | 72 | Accepted Hypothesis | 30–34 |
| Total configurations | 663,552 | Verification tasks | 94 |
| Tested configurations | 2,235 | Verification attempts | 210,090 |

based on hypotheses regarding verifiability and one function $\theta_{\text{eff}}$ based on hypotheses regarding verification effort. The overall score of option $o$ is then given by

$$\theta(o) = \gamma * \theta_{\text{ver}} + (1 - \gamma) * \theta_{\text{eff}}, \tag{1}$$

where $\gamma \in [0, 1]$ is a user-defined parameter to shift GUIDO's focus to either verifiability or verification effort. The score functions themselves are based on the *effect size* of the corresponding hypothesis tests.

Afterwards, a ranked list of configurations is generated by solving a *constrained optimization problem*, and the determined configuration is applied to verify the input program (5). If the predicted configuration is insufficient for the verification task, a *continuation mechanism* $\mathcal{M}$ is applied to compute the next promising configuration. A timing threshold $\tau$ marks the maximum time GUIDO spends on searching for and applying a configuration to a given verification task. The outcome is twofold: either (a) GUIDO finds a promising configuration in the given time span or (b) provides the list of the applied configurations for possible user inspection (6).

## III. Illustrative Application on the Deductive Program Verifier KeY

We applied GUIDO to the deductive program verifier KeY [2] in development version v2.7.0. KeY v2.7.0 comprises a total of 30 parameters, and each parameter is associated with *two* to *four* configuration options, of which exactly one has to be selected. In particular, checkboxes are encoded as parameters comprising two configuration options, namely `true` and `false`. We summarize the most important statistics of applying GUIDO to KeY in TABLE I, which we explain in more detail in the following.

We considered a total of 87 configuration options over the span of 30 control parameters. For instance, the example on hypotheses for KeY in Section II illustrates control parameter `One Step Simplification` with its configuration options `Disabled` and `Enabled`. Typically, numerous configuration options not only tune the verification algorithm itself, but also may change *what* is verified (e.g., absence of overflows, when integers are treated with the semantics of the programming language). Consequently, a number of options can be fixed in the beginning to already reduce the configuration space. As we only apply KeY to Java programs, we fixed options that would either prohibit a successful verification of such programs or would falsify the result. In total, we fixed 15 configuration options, such that 663,552 different configurations remain. To further reduce the configuration space, we used *three-wise* sampling (i.e., $t = 3$) employing the ICPL sampling algorithm [34], which is state-of-the-art for large configuration spaces [35]. As a result, 2,235 configurations remain. Thus, we performed a total of 210,090 verification attempts (i.e., each configuration sample is applied to each verification task) to compute the benchmark data set.

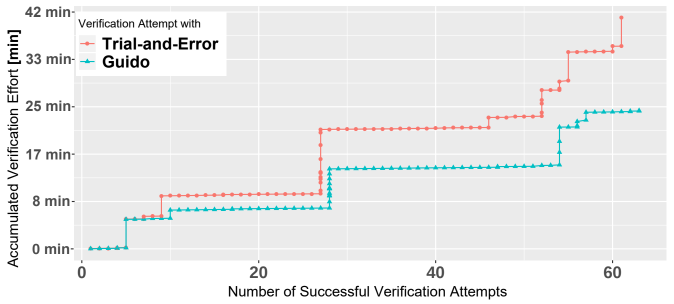


Fig. 3: Performed verification attempts and accumulated effort for GUIDO and a trial-and-error strategy using the program verifier KeY.

For deductive verification, our metric for verification effort is twofold: either (1) execution time of the verifier, or (2) the size (i.e., number of steps) of generated proofs. Finding configurations that lead to a reduced proof size may help in two ways. First, they are resource-beneficial in case of distribution, such as with proof-carrying code [36]. Second, as proofs can be replayed in re-verification attempts (e.g., when applied in continuous integration), smaller proofs are replayed faster and, thus, accumulate to less verification time [37], [38].

To capture our domain knowledge, we have formulated 72 hypotheses about verifiability and verification effort, and investigated each hypothesis as one independent experiment. We set our significance level to the commonly practiced 5% divided by the number of hypotheses (i.e., applying the Bonferroni-correction to mitigate the accumulated error). All null hypotheses with a p-value lower than this significance level $\alpha$ were rejected, meaning the alternative hypotheses were accepted. As illustrated in TABLE I, we accepted 30–34 hypotheses. The reason for this variation is that we randomly assigned each of the 94 tasks to exactly one of ten groups. To verify a task with GUIDO, we used all tasks of the remaining nine groups as training data (i.e., to evaluate our initial hypotheses). Hence, we evaluated our 72 defined hypotheses for each of the ten groups separately.

**Initial Results.** For an initial evaluation, we focus on the question: *How does the performance of* GUIDO *compare to a trial-and-error strategy for finding relevant configurations?*

In Fig. 3, we compare the performance of GUIDO to a trial-and-error strategy, which starts at the *default configuration* and continues to flip an option of a parameter randomly. In particular, configurations in GUIDO and the trial-and-error strategy are generated *until* (1) a case is proven successfully, (2) a timeout of five minutes is reached, or (3) the maximum number of unsuccessful verification attempts is exceeded. GUIDO applies a maximum of six configurations on a verification task. To enable a fair comparison, we also limited the number of attempts for the trial-and-error strategy to six. Each point on the lines in Fig. 3 represents a verification attempt. On the horizontal axis, we depict the number of closed proofs (i.e., successful verification attempts) for both strategies. On the vertical axis, we show the *accumulated* verification effort in minutes. With

63 verified tasks, GUIDO is able to close two more tasks than the trial-and-error strategy (i.e., 61 verified tasks). We did not depict the remaining 31 tasks, as none of these were closed in the given time limit. GUIDO needed approximately 26 minutes to run through all of them, whereas the trial-and-error strategy needed approximately 40 minutes, which is an increase of 63%.

**Effectiveness and Efficiency**

This experiment illustrates that GUIDO can be more effective for the deductive program verifier KEY than our baseline trial-and-error strategy, while also more efficient. GUIDO closed two more verification tasks while spending 63% less time overall.

## IV. Conclusion and Future Direction

Our on-going vision is to *mainstream* formal verification and help developers adopting formal method tools in software engineering practices. We follow this vision by focusing on configurable verification tools and suggest that tool builders implement means (e.g., GUIDO) to decrease the configuration burden. As a step towards that vision, we have presented GUIDO, a framework for the automatic configuration of deductive verification tools based on domain knowledge and statistical hypothesis testing. Although we acknowledge that techniques from the black-box machine learning context may outperform our chosen metric in the near future, we still believe that capturing domain knowledge and applying it actively in the configuration process increases accuracy while reducing the training data needed. Moreover, to the best of our knowledge, this is the first work that proposes a solution for automated configuration support in the context of deductive verification, and, as such, constitutes the current baseline.

Based on our experiments, we gained two main insights. First, as many hypotheses could not be accepted, proper knowledge about the influence of configuration options is hard to deduce without tool support. Second, the one-time-effort of formalizing expert knowledge and acquiring a benchmark data set (for *offline training*) is justified by an improvement in online performance, which is particularly important in the context of continuous integration and frequent re-verification.

For future work, we plan to extend our evaluation significantly by investigating more complex problems and also other program verifiers from the theorem proving and model checking domain.

## Acknowledgment

We thank Marlen Süwer, Markus Gamperle, and Carsten Pardylla for their contribution to the implementation.

## References

[1] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. MIT press, 2018.

[2] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, *Deductive software verification–The KeY book: from theory to practice*. Springer, 2016.

[3] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252.

[4] M. Gleirscher and D. Marmsoler, "Formal methods: Oversold? underused? A survey," *Computing Research Repository (CoRR)*, vol. abs/1812.08815, 2018. [Online]. Available: http://arxiv.org/abs/1812.08815

[5] M. Gleirscher, S. Foster, and J. Woodcock, "Assuring autonomous systems: opportunities for integrated formal methods?" *Computing Research Repository (CoRR)*, vol. abs/1812.10103, 2018. [Online]. Available: http://arxiv.org/abs/1812.10103

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter, "Specification and Verification: The Spec# Experience," *CACM*, vol. 54, pp. 81–91, Jun. 2011.

[7] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[8] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Proceedigs of the International Conference on Computer Aided Verification (CAV)*. Springer, 2011, pp. 184–190.

[9] P. Darke, S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla, "Veriabs: Verification by abstraction and test generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 457–462.

[10] C. Richter and H. Wehrheim, "Pesco: Predicting sequential combinations of verifiers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 229–233.

[11] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c." Springer, 2012, pp. 233–247.

[12] A. Knüppel, C. I. Pardylla, T. Thüm, and I. Schaefer, "Experience report on formally verifying parts of OpenJDK's API with KeY," in *Proceedings of the Fourth Workshop on Formal Integrated Development Environment*. Springer, 2018.

[13] A. Knüppel, T. Thüm, C. I. Pardylla, and I. Schaefer, "Understanding parameters of deductive verification: an empirical investigation of KeY," in *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*. Springer, 2018.

[14] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[15] T. Runge, T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson, "Comparing correctness-by-construction with post-hoc verification—a qualitative user study," in *International Symposium on Formal Methods*. Springer, 2019, pp. 388–405.

[16] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, vol. 20, no. 3-4, pp. 487–517, 2012.

[17] H. Ha and H. Zhang, "Deepperf: performance prediction for configurable software with deep sparse neural network," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Science, 2019, pp. 1095–1106.

[18] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering (EMSE)*, vol. 23, no. 3, pp. 1826–1867, 2018.

[19] G. T. Leavens and Y. Cheon, "Design by Contract with JML," Sep. 2006. [Online]. Available: http://www.jmlspecs.org/jmldbc.pdf

[20] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson, "Behavioral Interface Specification Languages," *CSUR*, vol. 44, no. 3, pp. 16:1–16:58, Jun. 2012.

[21] A. Grebhahn, N. Siegmund, S. Apel, S. Kuckuk, C. Schmitt, and H. Köstler, "Optimizing performance of stencil code with spl conqueror," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils)*, 2014, pp. 7–14.

[22] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[23] J. Kienzle, G. Mussbacher, P. Collet, and O. Alam, "Delaying decisions in variable concern hierarchies," in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, vol. 52, no. 3. ACM, 2016, pp. 93–103.

[24] R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside, "Modelling and multi-objective optimization of quality attributes in variability-rich software," in *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*. ACM, 2012, p. 2.

[25] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning (t)," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Science, 2015, pp. 365–373.

[26] S. De Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle, "Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case," in *Proceedings of the International Conference on Computer Aided Verification*. Springer, 2015, pp. 273–289.

[27] W. Mostowski, "Formalisation and verification of java card security properties in dynamic logic," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2005, pp. 357–371.

[28] ——, "Fully verified java card api reference implementation," *Verify*, vol. 259, pp. 136–151, 2007.

[29] W. Ahrendt, W. Mostowski, and G. Paganelli, "Real-time java api specifications for high coverage test generation," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 2012, pp. 145–154.

[30] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering (TSE)*, vol. 23, no. 5, pp. 279–295, 1997.

[31] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[32] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[33] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[34] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2012, pp. 46–55.

[35] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, "A classification of product sampling for software product lines," in *Proceedings of the International Software Product Line Conference (SPLC)*, vol. 1. ACM, 2018, pp. 1–13.

[36] G. C. Necula, "Proof-carrying code," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 1997, pp. 106–119.

[37] B. Beckert and V. Klebanov, "Proof reuse for deductive program verification," in *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Science, 2004, pp. 77–86.

[38] R. Hähnle, I. Schaefer, and R. Bubel, "Reuse in software verification by abstract method calls," in *Proceedings of the International Conference on Automated Deduction*. Springer, 2013, pp. 300–314.