Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY

Alexander Knüppel, Thomas Thüm, Carsten Immanuel Pardylla, and Ina Schaefer

TU Braunschweig
{a.knueppel,t.thuem,c.burmeister,i.schaefer}@tu-bs.de

Abstract. As formal verification of software systems is a complex task comprising many algorithms and heuristics, modern theorem provers offer numerous parameters that are to be selected by a user to control how a piece of software is verified. Evidently, the number of parameters even increases with each new release. One challenge is that default parameters are often insufficient to close proofs automatically and are not optimal in terms of verification effort. The verification phase becomes hardly accessible for non-experts, who typically must follow a time-consuming trial-and-error strategy to choose the right parameters for even trivial pieces of software. To aid users of deductive verification, we apply machine learning techniques to empirically investigate which parameters and combinations thereof impair or improve provability and verification effort. We exemplify our procedure on the deductive verification system KeY 2.6.1 and specified extracts of OpenJDK, and formulate 53 hypotheses of which only three have been rejected. We identified parameters that represent a trade-off between high provability and low verification effort, enabling the possibility to prioritize the selection of a parameter for either direction. Our insights give tool builders a better understanding of their control parameters and constitute a stepping stone towards automated deductive verification and better applicability of verification tools for non-experts.

Keywords: Deductive verification, Design by contract, Formal methods, Theorem proving, KeY, Control parameters, Automated reasoning

1 Introduction

Formal methods are intended to provide adequate solutions for software developers to rigorously *prove* that a piece of software is in line with a given specification [6, 9, 40, 41]. Besides light-weight methods intended to uncover the majority of defects early, such as code reviews and testing, there is need for advanced strategies to find the last defects. For instance, model checking is an automatic technique verifying that a given formal model (e.g., state machines) adheres to its specification [8, 39]. Although we expect our considerations to be more generally applicable to other formal verification techniques, we focus on deductive

verification, which is another technique that targets program verification directly on source code [1, 7, 2, 21, 43]. Essentially, an implementation together with its formal specification is translated into a logical formula and validity is proved by a theorem prover [43].

Despite considerable advances over the last decades and the advantage to be directly applied to source code, deductive verification is still only hesitantly applied in industrial software projects. Reasons are manifold. For example, there are doubts about the the cost-effectiveness of formal methods [29]. In particular, most legacy systems are not designed with formal verification in mind, which makes post-hoc specification and verification expensive for indutrial projects [4]. Moreover, developing sufficient formal specifications is error-prone and tedious [3, 2], and typically requires high expertise of the underlying proof theory. Even worse, full automation is not always possible because of the undecidability of the halting problem.

However, when full automation is feasible, a subsequent and often overlooked hurdle for inexperienced users is to parameterize the verification tool. Different implementations and specifications have different needs and modern verification tools provide parameters that are set by a user to control the verification process. For example, the parameter *loop treatment* can be used to decide whether loops are always unrolled or specified loop invariants are used. Consequently, successful verification often depends on those parameters.

Understanding all parameters requires a considerable amount of knowledge. Non-expert users may face problems when a piece of software cannot be verified even when implementation and specification are seemly correct. Furthermore, minimizing verification effort is important for industrial software. Complex software systems are frequently changed and formal specifications are adapted accordingly. In this process, past proof results may become invalid. A naive solution is then to follow a trial-and-error strategy by applying different parameter configurations, after which verification is restarted. This strategy, however, wastes a considerable amount of resources, making it less applied in industry.

We argue that a better understanding of parameterization allows tool builders to better support users of deductive verification. In particular, we investigate whether specific parameters have a larger influence on automated provability and whether specific options increase or decrease the verification effort. We focus on deductive verification following the *Design by Contract* paradigm [35]. Contracts are an extension to Hoare triples [23] and constitute a methodology to specify methods of imperative languages (i.e., Java or C) with *preconditions* and *postconditions*, and classes with *class invariants*. Callers of a contract-specified method have the obligation to fulfill the precondition and may therefore rely on the postcondition. Class invariants have to hold before and after method execution.

There exist numerous languages with support for contracts, such as Eiffel [36], Spec# [2], and the Java modeling language (JML) [31]. For the purpose of this paper, we analyze parameters of the state-of-the-art verification system KeY 2.6.1 [1]. KeY is a modern theorem prover with a large community intended to verify JML-specified Java programs. To empricially investigate KeY's parame-

ters, we formulate a total of 53 hypotheses in terms of provability and verification effort derived from the literature and documentation of KeY. Morever, we construct parameter-influence models based on our measurements to reason about which options influence the verification effort the most. To this end, we employ SPLConqueror [45], a framework which incorporates machine-learning techniques to measure the influence of parameters on non-functional properties. In summary, our contributions are the following.

- We formulate and empirically validate 53 hypotheses about parameterization in KeY, which provide clear recommendations for users who aim to verify pieces of software automatically.
- We empirically evaluate the influence of KeY's parameters with respect to provability and verification effort with machine learning.
- We identify parameters that depict a trade-off between higher provability and lower verification effort and discuss consequences for users and tool builders.

2 Problem Statement

With formal verification, our goal is to identify the last remaining defects. When automatic software verification fails, users are confronted with a diverse set of reasons. Typically, most common reasons consist of (a) a wrong implementation, (b) a wrong or insufficient specification (e.g., loop invariants are missing or too weak), (c) insufficient heuristics of the verification tool (e.g., when automatically inferring loop invariants or instantiating quantifiers), or (d) the verification task times out after the maximum number of proof steps or heap memory is exceeded.

As if these hurdles are not enough, a subsequent challenge is that parameterization has also a great effect on the outcome and the default values are oftentimes not sufficient. Getting the parameters right from the beginning makes deductive verification significantly more successful and cost-effective.

We divide parameters of deductive verification broadly into two categories. The first category describes qualitative parameters that explicitly change *what to prove*. For instance, there is an option in KeY to ignore *integer overflows*. Consequently, implementations that cause an integer overflow are not verifiable with this setting. Depending on the context and how the implementation is facilitated, however, verifying the absence of integer overflows is crucial. Those parameters must be set by users or have at least a well-chosen default value.

The second category describes parameters that only influence provability and verification effort (i.e., *how to prove*). For instance, there is a parameter in KeY for method call treatment; a method call is either always replaced with an existing contract (i.e., contracting), or its implementation is always inlined (i.e., method expand). Typically, contracting is faster and results in lower verification effort. However, in case of missing or insufficient contracts, a method can only be proved correct with method inlining.

To verify a piece of software automatically, a user must first identify what to prove and has to set respective parameters accordingly (e.g., enabling detection of integer overflows). In a second step, a user typically starts the verification process

```
/*@ public normal_behavior
  @ requires T > 0;
  @ ensures \result < T;
  @*/
public /*@ pure @*/ int modT(int input, int T) {
  return input % T;
}
```

Listing 1. Method Computing the Modulo of Integer Values

```
/*@ public normal_behavior
  @ requires e != null;
  @ ensures contains(e);
  @ ensures collectionSize == \old(collectionSize) + 1;
  @ ensures \result;
  @ assignable elements;
  @*/
boolean add(/*@nullable@*/ Object e);
```

Listing 2. Method ArrayList.add(Object) Specified with Contracts in JML

with default parameters or the last used configuration to check whether they suffice. In case of failure, oftentimes parameterization is changed and verification is restarted in a trial-and-error manner. Moreover, as frequent changes to software systems are the common case, reducing the verification effort is another important requirement. Hence, having a better understanding of the control parameters and providing better tool support would tremendously help inexperienced users to apply deductive verification more successfully.

In the following, we depict two examples, where default parameters are either insufficient or result in an increased verification effort. In Listing 1, we illustrate a small example of a formally specified method modT(int, int) that gets two integer values as input and computes the modulo between both. The precondition is denoted by keyword requires and states that input parameter T must be greater than 0. The postcondition is denoted by keyword ensures and states that the return value will always be less than T given the precondition. Keyword \result represents the return value. Notably, method modT(int, int) is not automatically verifiable with KeY's default parameters. As the example is small, implementation and specification are readily comprehensible and seemly fit together. In particular, the reason is a parameter called *Arithmetic treatment*. The set of possible options is {Basic, DefOps, Model Search}, where Basic is the default value. However, unlike DefOps, Basic is incapable of evaluating the modulo operator. Starting from the default parameters, choosing DefOps as value for *Arithmetic treatment* suffices to verify method modT(int, int). Although it is possible to modify the postcondition to \result== input%T and verify it indeed with the default parameters, such modifications are hard to find for real-world software systems.

In Listing 2, we depict another example, where we specified method add(Object) of class ArrayList in JML. Class ArrayList is part of the Collection-API and imple-

ments the interface Collection. The precondition states that callers of add(Object) can only rely on the postcondition when they provide an instantiated object. The postcondition states that (a) the input object is indeed part of the list after successful method execution, (b) the list's size is incremented by one, and (c) the return value is true. In a postcondition, keyword \old evaluates the expression before method execution. Keyword assignable represents the framing condition (i.e., a set of locations). Implicitly, locations excluded from the frame are not allowed to be modified. The example also contains *queries*, which are side-effect free methods that can be called in specifications (e.g., method contains(Object)). Internally, method add(Object) calls method ensureCapacity(Object), which increases the capacity of the respective list by one, if necessary. While the example depicts a trivial contract, the verification effort with KeY's default parameters can be reduced from 33,748 proof steps to 13,328 proof steps when changing the parameter *Quantifier treatment* from No Splits with Progs to No Splits.

Ideally, with assistance of extended tooling, a developer is capable of understanding the influence of various control parameters on provability and verification effort. For instance, a recommendation system may suggest to change the option of parameter *Arithmetic treatment* for the example depicted in Listing 1, as only after carefully studying the tool tips in KeY it becomes apparent that *Arithmetic treatment*::Basic cannot evaluate the modulo operator.

3 Parameters of Deductive Verification with KeY

As already mentioned, KeY provides numerous parameters to control *what* and *how* a piece of software is verified. In particular, there exist three categories of parameters in KeY, namely *search strategy options*, *taclet options*, and *general options*. Search strategy options control to what extent and in which order KeY applies inference rules to automatically verify a method. Taclet options control rather *what* to prove (e.g., integer overflow) and, thus, are typically fixed for a verification target. General options enable the employment of an SMT solver or allow for one step simplification, which combines single inference rules into one.

Although KeY's automated proof strategy algorithm is configurable, the difficulty here is that it has grown over many years, with participation of many different researchers and universities. The effect of some parameters on provability and verification effort is therefore challenging to anticipate. We aim to provide a better understanding on how specific parameters improve or impair provability and verification effort. Based on our experience and observations combined with studying the online documentation[1], numerous publications [13, 15, 17, 18, 26, 27, 32, 42], all tool tips in KeY, and the KeY book [1], we formulated a total of 38 assumptions that we empirically evaluate by deriving 51 statistical hypotheses in the next section. Related to Listing 1 and Listing 2, two examples for assumptions about the parameters *Arithmetic treatment* and *Quantifier treatment* with respect to provability and verification effort are the following.

[1] `https://www.key-project.org/applications/program-verification/` and `http://i12www.ira.uka.de/key/download/quicktour/quicktour-2.0.zip`

Assumption 19 (Arithmetic Treatment - Basic and DefOps) *If a specification case is provable with option* Basic*, it is also provable with option* DefOps.

Assumption 23 (Quantifier Treatment - Verification Effort) *The verification effort with option* Free *is at least as great as with option* No Splits with Progs. *The verification effort with option* No Splits with Progs *is at least as great as with option* No Splits. *The verification effort with option* No Splits *is at least as great as with option* None.

In Table 1, we give an overview on all our assumptions in a short from. The first column denoted by *Assumption* represents an identifier for the respective assumption and the second column denoted by *Parameter* represents the parameter which is subject to the assumption. For our significance tests, we only relate two values of a parameter with each other (i.e., fourth and fifth column). That is why there exist more experiments (i.e., third column) than assumptions. For instance, Asumption 23 relates four values with each other in terms of verification effort. The type of the *depended variable* is represented in the sixth column, where provability is denoted by P and verification effort is denoted by VE. In particular, for reasoning about the verification effort we always assume that a verification target is provable with both subjected options. The last column represents whether one option improves, impairs, or does not influence the outcome. For provability, $O_a \leq O_b$ means that O_b provides a higher chance of provability that O_a. For verification effort, $O_a \leq O_b$ means than O_b leads to a greater effort than O_a. The opposite meaning for each depended variable is denoted by $\geq$ and no significant influence is denoted by $<>$.

Assumption	Parameter	Hypothesis	First option	Second option	Requirement	Dependency
1	Stop At	1	Default	Unclosable	P	$<>$
2	Stop At	2	Default	Unclosable	VE	$<>$
3	One Step Simplification	3	Enabled	Disabled	P	$<>$
4	One Step Simplification	4	Enabled	Disabled	VE	$\leq$
5	Proof Splitting	5	Delayed	Free	P	$\leq$
6	Proof Splitting	6	Delayed	Free	VE	$\leq$
7	Proof Splitting	7	Off	Free	P	$\leq$
		8	Off	Delayed	P	$\leq$
8	Proof Splitting	9	Off	Free	VE	$\leq$
		10	Off	Delayed	VE	$\leq$
9	Loop Treatment	11	Invariant	Loop Scope Invariant	P	$\leq$
10	Loop Treatment	12	Invariant	Loop Scope Invariant	VE	$\geq$
11	Dependency Contracts without accessible-Clauses	13	On	Off	P	$<>$
12	Dependency Contracts without accessible-Clauses	14	On	Off	VE	$<>$
13	Query Treatment without queries	15	On	Restricted	P	$<>$
		16	On	Off	P	$<>$
14	Query Treatment without queries	17	On	Restricted	VE	$<>$
		18	On	Off	VE	$<>$
15	Query Treatment	19	Off	Restricted	P	$\leq$
		20	Resticted	On	P	$\leq$

16	Query Treatment	21	Restricted	On	VE	<>
17	Expand Local Queries	22	On	Off	VE	≥
18	Expand Local Queries	23	On	Off	P	≥
19	Arithmetic Treatment	24	Basic	DefOps	P	≤
20	Arithmetic Treatment	25	DefOps	ModelSearch	P	<>
21	Quantifier Treatment without Quantifiers	26	None	No Splits	P	<>
		27	None	No Splits With Progs	P	<>
		28	None	Free	P	<>
22	Quantifier Treatment without Quantifiers	29	None	No Splits	VE	<>
		30	None	No Splits With Progs	VE	<>
		31	None	Free	VE	<>
23	Quantifier Treatment	32	None	No Splits	P	≤
		33	No Splits	No Splits With Progs	P	≤
		34	No Splits With Progs	Free	P	≤
24	Quantifier Treatment	35	Free	No Splits With Progs	VE	≥
		36	No Splits With Progs	No Splits	VE	≥
		37	No Splits	None	VE	≥
25	Class Axiom Rule without Axioms	38	Free	Delayed	P	<>
		39	Free	Off	P	<>
26	Class Axiom Rule without Axioms	40	Free	Delayed	VE	<>
		41	Free	Off	VE	<>
27	Class Axiom Rule	52	Considered separately			
28	Class Axiom Rule	53	Off	Delayed	P	<>
29	Strings	42	On	Off	P	≥
30	Strings	43	On	Off	VE	<>
31	BigInt	44	On	Off	P	≥
32	BigInt	45	On	Off	VE	<>
33	IntegerSimplificationRules	46	Full	Minimal	P	≥
34	IntegerSimplificationRules	47	Full	Minimal	VE	<>
35	Sequences	48	On	Off	P	≥
36	Sequences	49	On	Off	VE	<>
37	MoreSeqRules	50	On	Off	P	≥
38	MoreSeqRules	51	On	Off	VE	<>

Table 1: Investigated Parameters and Formulated Hypotheses

We considered Assumption 27 separately. The reason is that this assumption does not compare options, but states that specification cases are not provable based on particular conditions:

Assumption 27 (Class Axiom Rule) *If a method writes onto a location on the heap and there exists at least one class invariant that refers to this location, then option* Off *is not sufficient to verify this method.*

To summarize, we formulated a total of 38 assumptions on 47% of all available parameters in KeY. In Figure 1, we depict all assumptions about parameters for which we identified an order of their options with respect to higher and lower

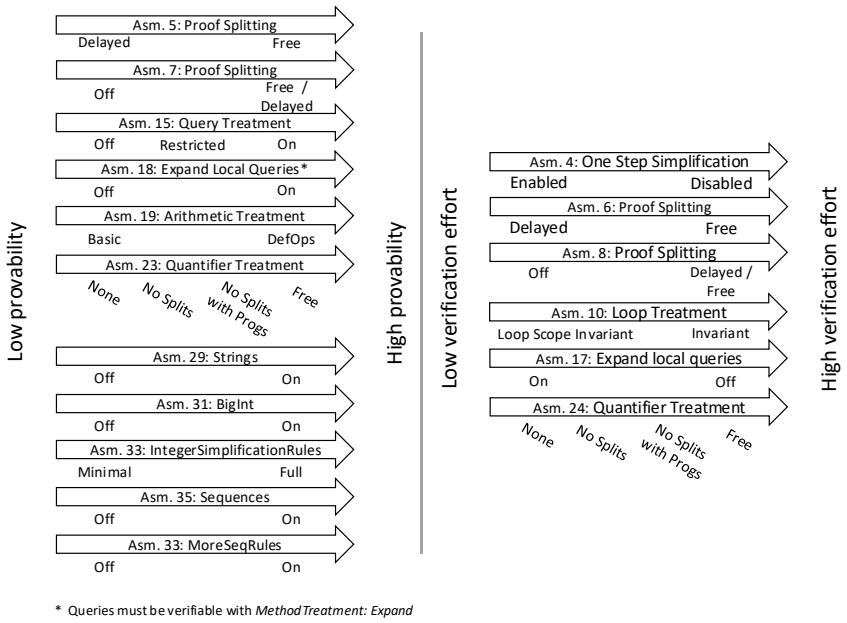

Fig. 1. Overview on Assumptions for Provability and Verification Effort

provability and verification effort. Confirmation of these assumptions helps in prioritizing parameters for fine-tuning in terms of provability and verification effort.

4 Empirical Evaluation of KeY's Parameters

In a series of experiments, we evaluate the assumptions that we formulated in the last section. All formulated assumptions, evaluation artifacts, results, and the verification target can be found online.[2]

4.1 Experimental Setup

For our verification target, we formally specified parts of OpenJDK's Collection-API with JML. Reasons to focus on OpenJDK are threefold, namely (a) it represents a widespread and highly applied real-world software, (b) there exists already an informal specification in the JavaDocs comments that we utilize for our formal specification, and (c) it is open-source and the only Java distribution that allows us to add contracts and freely distribute it. A method can have more than one contract (e.g., when different preconditions are connected with distinct postconditions), which we refer to as *specification cases* in the following. In total, our test study comprises 27 specification cases distributed over the interface Collection and classes ArrayList, LinkList, Arrays, and Math, which we specified in

[2] http://github.com/AlexanderKnueppel/UnderstandingParametersInKeY

a complementary study [30]. To reduce bias in our experiments, we only included specification cases that can be verified automatically with at least one parameter configuration.

All assumptions refer either to *provability* or *verification effort*. For provability, the dependent variable is whether a proof can be found automatically or not. For verification effort, the dependent variable is the number of proof steps. Independent variables are in both cases the current parameterization and the specification cases.

While evaluating our assumptions on all possible parameter configurations yields the most accurate result, it is impractical due to the combinatorial explosion in the number of options. In total, there exist 1,990,656 valid parameter configurations. To find a reasonable and meaningful number of configurations, we apply pairwise interaction sampling [11], which requires that every pair of options of different parameters is present at least once in the set of parameter configurations. In essence, 1,084 configurations suffice. For assumptions that compare at least two options related to the verification effort, we apply the non-parametric paired Wilcoxon-Test [50]. The rationale for a non-parametric test is that we cannot expect the distribution of the proof effort to be normal. For assumptions that only consider one option (i.e., Assumption 27), we apply a 1-sample Wilcoxon-Test [50]. For assumptions that compare at least two values related to provability, we apply a McNemar-Test [34]. For each experiment, we define a significance level of 5% and we set the the maximal number of proof nodes to 500,000, after which a verification task times out.

4.2 Empricial Evaluation of Assumptions

In Table 2, we summarize all experiments together with their statistical hypotheses, respective p-value, and outcome. Depending on the statistical test and formulation of the assumption, we need to define and evaluate different kind of statistical hypotheses. If the assumption states that *there is no significant difference*, we cannot formulate a null hypothesis H_0 that we would like to reject in favor of the assumption. In this case, we use the assumption itself as the null hypothesis and denote the *hypothesis type* as H_0. The consequence is that we can only reject or not reject our assumption, but never accept it. The preferred outcome is *not rejected*, as otherwise our assumption would be indeed wrong. If we can formulate a null hypothesis, we use the hypothesis type $H_{\leq}$ and $H_{\geq}$ for assumptions that paraphrase *at least* or *at most* relationships between options (e.g., Assumption 23) or simply H_A otherwise. In this case, the preferred outcome is *accepted*. The result is always *not rejected*, if we could not reject the null hypothesis.

Parameter	Assumption	Experiment	Hypothesis Type	p-value	Result
Stop At	1	1	H_0	NA	not rejected
	2	2	H_0	$2,488 * 10^{-2}$	**rejected**
One Step Simplification	3	3	H_0	NA	not rejected
	4	4	H_A	$< 2,2 * 10^{-16}$	**accepted**
Proof Splitting	5	5	$H_{\leq}$	NA	not rejected
	6	6	H_A	$7,7 * 10^{-9}$	**accepted**

Proof Splitting	7	7	$H_{\leq}$	$3,252 * 10^{-9}$	**accepted***
		8	$H_{\leq}$	$3,252 * 10^{-9}$	**accepted***
	8	9	H_A	$4,147 * 10^{-2}$	**accepted**
		10	H_A	$6,063 * 10^{-1}$	not rejected
Loop Treatment	9	11	H_0	NA	not rejected
	10	12	H_A	$9,186 * 10^{-3}$	**accepted**
Dependency Contracts	11	13	H_0	NA	not rejected
	12	14	H_0	1	not rejected
Query Treatment	13	15	H_0	NA	not rejected
		16	H_0	NA	not rejected
	14	17	H_0	$1,422 * 10^{-1}$	not rejected
		18	H_0	1	not rejected
	15	19	$H_{\leq}$	$1,573 * 10^{-1}$	not rejected
		20	$H_{\leq}$	NA	not rejected
	16	21	H_A	$1,706 * 10^{-1}$	not rejected
Expand Local Queries	17	22	H_A	$6,601 * 10^{-2}$	not rejected
	18	23	$H_{\leq}$	$4,55 * 10^{-2}$	**accepted***
Arithmetic Treatment	19	24	$H_{\geq}$	$9,237 * 10^{-13}$	**accepted***
	20	25	H_A	$3,173 * 10^{-1}$	not rejected
Quantifier Treatment	21	26–28	-	-	**rejected**
	22	29–31	-	-	**rejected**
	23	32	$H_{\leq}$	$4,55 * 10^{-2}$	**accepted***
		33	$H_{\leq}$	$1,573 * 10^{-1}$	not rejected
		34	$H_{\leq}$	NA	not rejected
	24	35	H_A	$7,186 * 10^{-1}$	not rejected
		36	H_A	$2,869 * 10^{-1}$	not rejected
		37	H_A	$1,562 * 10^{-1}$	not rejected
Class Axiom Rules	25	38	H_0	NA	not rejected
		39	H_0	NA	not rejected
	26	40	H_0	NA	not rejected
		41	H_0	NA	not rejected
	27	52	H_0	NA	not rejected
	28	53	H_0	NA	not rejected
Strings	29	42	$H_{\geq}$	NA	not rejected
	30	43	H_0	1	not rejected
BigInt	31	44	$H_{\geq}$	NA	not rejected
	32	45	H_0	1	not rejected
IntegerSimplificationRules	33	46	$H_{\geq}$	$5,32 * 10^{-4}$	**accepted***
	34	47	H_0	$8,783 * 10^{-1}$	not rejected
Sequences	35	48	$H_{\geq}$	NA	not rejected
	36	49	H_0	$2,61 * 10^{-1}$	not rejected
MoreSeqRules	37	50	$H_{\geq}$	NA	**accepted***
	38	51	H_0	$3,458 * 10^{-1}$	not rejected

* after manual inspection

Table 2: Experimental Results with Accepted and Rejected Assumptions

Based on our results, we had to neglect two hypotheses, namely Assumption 21 and Assumption 22 stating that *Quantifier treatment* does not effect provability and verification effort if the logical formula under verification is quantifier-free. We discovered that each verification of a program in KeY works with quantification internally as soon as assignable-clauses are used, even when no quantifiers are used in the contracts. Furthermore, we had to reject Assumption 2 (i.e., parameter

Hypothesis 7		Off	
Proof Splitting		Closed	Open
Free	Closed	114	37
	Open	0	142

Hypothesis 24		Basic	
Query Treatment		Closed	Open
DefOps	Closed	119	51
	Open	0	164

Hypothesis 8		Off	
Proof Splitting		Closed	Open
Delayed	Closed	116	37
	Open	0	143

Hypothesis 32		None	
Quantifier Treatment		Closed	Open
No Splits	Closed	132	4
	Open	0	107

Hypothesis 23		On	
Expand local queries		Closed	Open
Off	Closed	82	0
	Open	4	105

Hypothesis 46		Full	
IntegerSimplificationRules		Closed	Open
Minimal	Closed	122	0
	Open	12	151

Table 3. Contingency Tables of Manually Inspected Assumptions

Stop at does not influence the verification effort), as the statistical result was significant (p-value: $2,488 * 10^{-2}$).

Four hypotheses about how the verification effort is influenced were accepted (i.e., Hypothesis 4, 6, 9, and 12). Six additional assumptions about how provability is influenced were accepted after an additional manual inspection (Hypothesis 7, 8, 23, 24, 32, 46). The reason for manual inspection is that the employed McNemar-Test is always two-sided. This means that the direction of difference (i.e., positive or negative) is not directly apparent. For the manual inspection, we use contingency tables to decide whether the null hypothesis can indeed be rejected. In Table 3, we depict the significant hypotheses, which were tested in the McNemar-Test, with their contingency tables. A hypothesis can be accepted if the sum of the first row is unequal to the sum of the first column, and analogously for the second row and second column. *Closed* and *Open* refer to whether a verification task was solved automatically or not. For instance, for Hypothesis 7, a total of 114 verification tasks of all verification tasks performed were solved automatically with *Proof splitting*::Free and *Proof splitting*::Off, whereas 142 verification tasks could not be solved automatically with either option.

One oddity is Assumption 37. 100% of the data correlates with its statement, which is why we could not compute the p-value but accepted the hypothesis nonetheless. In summary, three hypotheses were rejected and eleven hypotheses were accepted. The remaining hypotheses could neither be rejected nor confirmed and may have to be investigated in more detail and with additional verification targets in future studies.

4.3 Learning a Parameter-Influence Model

Our assumptions only state which options do have or do not have an effect on provability or verification effort. However, in terms of verification effort, it is also interesting to know which options have a larger effect than others. While previous assumptions help to exclude some options when optimizing the verification effort, we are even interested in prioritizing options according to their impact.

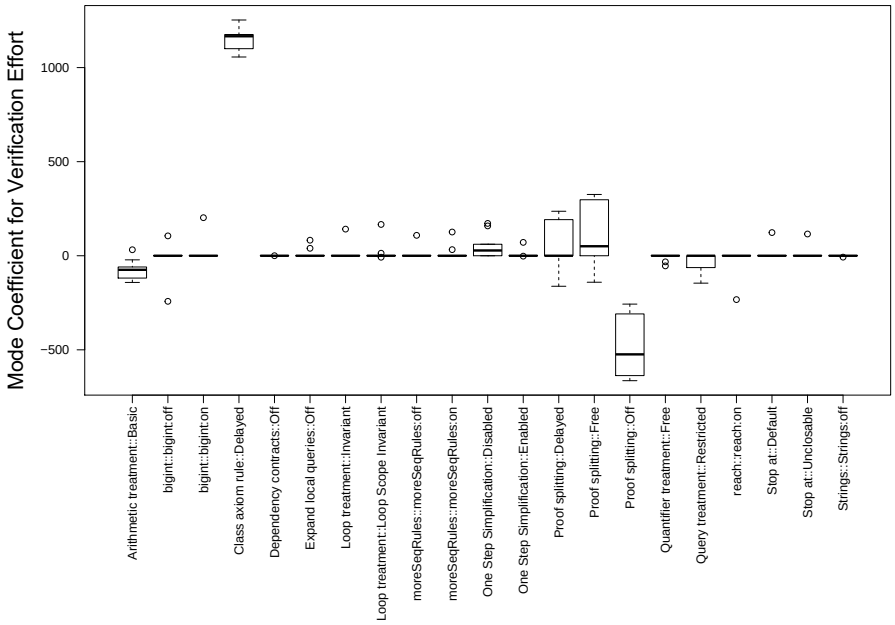

Fig. 2. Influence of Various Options on Verification Effort

Based on our experiments, we collected a considerable amount of data points, which depict the verification effort with respect to different options. In a nutshell, we decided to learn a parameter-influence model based on our data to draw these relevant conclusions about which options have a larger impact on the verification effort. To derive a general model, we use all of our specification cases as input for machine-learning techniques. To this end, we employ SPLConqueror [45], a framework which incorporates machine-learning techniques to measure the influence of parameters on non-functional properties.

Our samples comprise only specification cases that are automatically verified, as the number of proof steps in unclosed proofs is not meaningful. Moreover, to increase our confidence in the performance-influence model, we applied cross-validation to learn a total of ten models. To this end, we partitioned all verified specification cases into ten subsets and used nine randomly chosen subsets to train each of the ten models.

In Figure 2, we depict the results of our ten prediction models using boxplots that relate numerous options with verification effort. Each boxplot presents a factor that indicates whether an option improves (negative value) or impairs (positive value) the verification effort. Notably, there exist numerous options that were discarded in the training process of all ten models, as too few verification tasks could be closed with them automatically. Therefore, we also omitted them in Figure 2.

For most options, the median is close to zero, which is why we cannot reason about their influence. Exceptions are *Arithmetic treatment*::Basic, *Class axiom*::Delayed, *One Step Simplification*::Disabled, *Proof splitting*::Delayed, *Proof*

splitting::Free, *Proof splitting*::Off, and *Query treatment*::Restricted. With respect to the median, we achieve the largest improvement with *Proof splitting*::Off followed by *Arithmetic treatment*::Basic. *Class axiom*::Delayed leads to the largest deterioration of the verification effort.

While we did not make any assumptions about options *Arithmetic treatment*::Basic and *Class axiom*::Delayed with respect to verification effort, Figure 2 reveals that *Arithmetic treatment*::Basic almost always reduces the effort, whereas *Class axiom*::Delayed always and significantly impairs it. Hence, we can derive two new assumptions from our parameter-influence models. Nonetheless, these assumptions are not derived from the literature, but based on our exploratory study and have to be evaluated in future studies.

To briefly summarize, we identified seven options that reasonably impact the verification effort. Our results allow us to prioritize these options when provability is already ensured. However, we also discovered that *Arithmetic treatment*::Basic only insignificantly reduces the verification effort based on our specified extract of OpenJDK and we previously confirmed that what is provable with *Arithmetic treatment*::Basic is also provable with *Arithmetic treatment*::DefOps (cf. Assumption 19). Hence, *Arithmetic treatment*::DefOps may be the better choice for all verification tasks. Notably, numerous of our assumptions coincides with the models' predictions (i.e., Assumption 4, 6, 8, and 16).

4.4 Threats to Validity

The measured verification effort may not be representative, as we decided to count the proof nodes that KeY produces internally, which may vary in complexity and execution time. One alternative is to measure the overall execution time needed to verify a method. We decided against it, as execution timing depends on numerous external factors, such as computing power, parallel processes, and even the currently active virtual machine, whereas the number of proof nodes is a reproducible measurement.

Our verification target (i.e., OpenJDK's Collection API) may not comprise enough representative specification cases, as we did not specify many loop invariants or complex algorithms. Nevertheless, we specified real-world Java code, for which the specification effort was already tremendously high (i.e., it took us numerous iterations and months to be amenable for automatic verification). For all employed specification cases there exist at least one parameter configuration, which suffices to automatically verify it. Moreover, we computed all results on high-end servers over a period of two months. Specifying and verifying more specification cases would take considerably longer.

We only formulated assumptions about parameters that are used in KeY 2.6.1. It is thus questionable whether our considerations can be generalized to other verification systems. However, parameterization for non-expert users is also challenging for other techniques and tools, such as model checking with Java Pathfinder [46]. Furthermore, the chosen dependent variables (i.e., provability and verification effort) are typically most meaningful for users and tool builders of other verification systems, too.

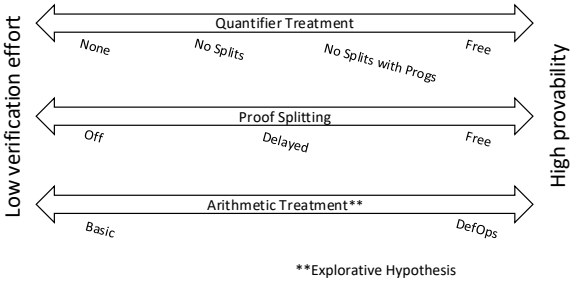

Fig. 3. Trade-off between Lower Verification Effort and Higher Provability

5 Suggestions for Users and Tool Builders

Our assumptions depict numerous options that a user should prioritize for tuning to increase provability and verification effort. For instance, parameters that need to be changed (i.e., differ from the default option) are *Quantifier Treatment* and *Proof splitting*, which can be set to Free, and *Arithmetic Treatment*, which can be set to DefOps. Moreover, *Stop At* should stay at Default and *Expand Local Queries* should stay at On.

If provability is ensured, verification effort can be tweaked. Our hypotheses state that *One Step Simplification* should be set to free,

Proof Splitting should be set to off, and *Loop Treatment* to Loop Scope Invariant. In particular, the parameter-influence models illustrated that *Proof Splitting::*Off decreases verification effort the most compared to all options. Moreover, the models revealed that *Arithmetic Treatment* and *Class Axiom Rule* have also an impact. *Arithmetic Treatment::*Basic is preferred to *Arithmetic Treatment::*DefOps, whereas *Class Axiom Rule::*Delayed should be avoided when possible. Nevertheless, our suggestion is to always start with *Arithmetic Treatment::*DefOps, as it is often needed for provability and the gain in terms of verification effort seems to be insignificant.

Based on our results, we can also identify parameters, whose options represent a trade-off between higher provability and lower verification effort. In essence, these parameters are *Quantifier treatment*, *Proof splitting*, and *Arithmetic treatment*, which are illustrated in Figure 3. Whenever provability is ensured, these options allow a user to decrease the verification effort.

Derived from our assumptions, some options have no measured impact on the verification effort but influence provability. It is thus questionable why a user is confronted with these options. A solution would be to provide different modes for different requirements, such as a *simple view* for inexperienced users that hides specific options. In particular, such a view may discard parameters *BigInt*, *IntegerSimplificationRules*, *Sequences*, *MoreSeqRules*, *One Step Simplification*, and *Stop At* for the mentioned reason.

Another suggestion for tool builders is to implement a recommendation system for parameterization, which enhances user experience. KeY could provide

hints to users to increase provability if a method cannot be verified. For instance, option *Proof Splitting::*Off may replace option *Proof Splitting::*Free or option *IntegerSimplificationRules::*Minimal may replace option *IntegerSimplificationRules::*Full. Moreover, KeY could try to automatically fine-tune parameters during verification based on the very same technique.

6 Related Work

A survey on different languages for behavioral contracts was done by Hatcliff et al. [21]. Besides KeY with its specialization on Java source code, there exist alternative tools for deductive program verification of other languages, such as Spec# [2], VCC [10] for verifying concurrent C, and the Why platform [49], which comprises tools for the verification of WhyML, Java, and C programs [33, 16, 12]. For the purpose of this paper, we concentrated on KeY, as (a) it provides numerous parameters, (b) it has an active community, and (c) we already gained ample and practical experiences with it [24, 46–48].

Gouw et al. [14] investigated the correctness of OpenJDK's TimSort with KeY and discovered an exploitable bug in its implementation. They changed parameterization even *during* the search for proofs, which is difficult as it requires to find meaningful interruption points. This is an indicator that an advanced understanding of the parameters is indispensable to verify real-world software with deductive verification.

Another formal verification technique requiring an understanding about its parameters is model checking. SPIN [25] is a software model checker that focuses on finite state machines and provides numerous configurable options and optimizations, such as partial order reduction, state compressions, and bitstate hashing. Java Pathfinder (JPF) [22] is a software model checker focusing on Java source code. JPF can be parameterized and extended in a variety of ways and is build upon a general and uniform configuration management. Configuring JPF for *efficiently* finding defects for a given verification task needs a considerable amount of knowledge about model checking.

Optimizing the selection of parameters of configurable programs is a widely researched area. Benavides et al. [5] analyzed the performance of CSP, SAT, and BDD solvers in finding a valid configuration. Ochoa et al. [37] transform a set of configurations into a CSP solver to find a non-conflicting set of configurations that adhere to particular business objectives, such as costs, time, and human resources, of multiple stakeholders. Siegmund et al. [45] proposed SPL Conqueror, which we used to learn our prediction models. Despite its initial connection to software product lines, SPL Conqueror is used by various researchers to learn models that predict the influence of non-functional properties in configurable software [19, 20, 28, 38, 44]. We provide an additional use case for SPL Conqueror, as we learned parameter-influence models to argue about how the selection of particular options influence the verification effort.

7 Conclusion

Our long-term goal is to make deductive verification accessible for mainstream software developers. Although formal methods improved significantly over the last decades, software developers still struggle to specify and verify even trivial pieces of software. One often overlooked hurdle that inexperienced users face is parameterization. While parameterization of formal method tools comes with the promise to ease the process of automatic verification, we exhibited that setting the right values for the ever growing amount of parameters is challenging.

In particular, our focus is on parameters of deductive verification, where we used the verification system KeY 2.6.1 as an example.

We formulated a total of 38 assumptions how options in KeY improve or impair provability and verification effort. We derived a total of 51 statistical hypotheses and empirically measured the effect of different parameter configurations by employing significance tests and machine-learning techniques.

Our empirical investigation is a stepping stone towards automated deductive verification and better applicability for non-experts. Only three of our initial assumptions had been invalidated. We identified options that should be prioritized according to their impact on verification effort when provability is ensured. Moreover, we identified three parameters (i.e., *Quantifier Treatment*, *Proof Splitting*, and *Arithmetic Treatment*), whose options represent a trade-off between provability and verification effort. Our insights provide valuable recommendations to users on which parameters to prioritize given a verification requirement. Moreover, tool builders can utilize our insights to improve on the user experience. For instance, implementing a recommendation system for parameters based on our investigation would help users to verify software more easily. Furthermore, KeY may hide insignificant parameters in specific verification scenarios or fine-tune parameters automatically during proofs.

For future work, it is necessary to employ more verification targets to investigate the assumptions that could not be accepted. Moreover, it would be interesting to implement a system for parameter recommendations that provides even more fine-grained recommendations based on contracts and methods. Furthermore, we only measured the effect of single values of parameters on provability and verification effort. However, specific values of different parameters may interact with each other and therefore have a larger or even reversed effect when used in combination. Finally, investigating parameterization of other verification tools is indispensable to help more industrial software developers to integrate formal methods in their everyday software development tasks.

Acknowledgments. This work was supported by the DFG (German Research Foundation) under the Researcher Unit FOR1800: Controlling Concurrent Change (CCC). We acknowledge Richard Bubel, Reiner Hähnle, Dominik Steinhöfel, Norber Siegmund, Alexander Grebhahn, Christian Kästner, Sven Apel, and Stefan Krüger for fruitful discussion and valuable feedback throughout this work. We also thank all reviewers for their valuable feedback and corrections.

References

1. 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.
2. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. *Communications of the ACM*, 54:81–91, June 2011.
3. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons Learned from Microkernel Verification–Specification is the new Bottleneck. *arXiv preprint arXiv:1211.6186*, 2012.
4. B. Beckert, T. Bormer, and D. Grahl. Deductive Verification of Legacy Code. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, pages 749–765. Springer, 2016.
5. D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 677–682, 2005.
6. J. Bowen and V. Stavridou. Safety-critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, 1993.
7. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. 7(3):212–232, June 2005.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachussetts, 1999.
9. E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.
10. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.
11. M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-configurable Systems in the Presence of Constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 129–139. ACM, 2007.
12. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.
13. Á. Darvas, F. Mehta, and A. Rudich. Efficient Well-Definedness Checking. In *Proceedings of the International Joint Conference on Automated Reasoning*, pages 100–115. Springer, 2008.
14. S. De Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. OpenJDKs Java.utils.Collection.sort() is Broken: The Good, the Bad and the Worst Case. In *International Conference on Computer Aided Verification*, pages 273–289. Springer, 2015.
15. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
16. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification*, pages 173–177. Springer, 2007.
17. C. D. Gladisch. Model Generation for Quantified Formulas with Application to Test Data Generation. *Proceedings of the International Journal on Software Tools for Technology Transfer*, 14(4):439–459, 2012.

18. J. Gosling. *The Java Language Specification*. Addison-Wesley Professional, 2000.
19. 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)*, pages 7–14, 2014.
20. J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 301–311. IEEE Press, 2013.
21. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. 44(3):16:1–16:58, June 2012.
22. K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. 2(4):366–381, 2000.
23. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.
24. S. Holthusen, M. Nieke, T. Thüm, and I. Schaefer. Proof-Carrying Apps: Contract-Based Deployment-Time Verification. Springer, Oct. 2016. To appear.
25. G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, 1997.
26. E. Hubbers and E. Poll. Reasoning About Card Tears and Transactions in JAVA CARD. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 114–128. Springer, 2004.
27. M. Huisman and W. Mostowski. A Symbolic Approach to Permission Accounting for Concurrent Reasoning. In *Proceedings of the Parallel and Distributed Computing (ISPDC), 2015 14th International Symposium on*, pages 165–174. IEEE, 2015.
28. J. Kienzle, G. Mussbacher, P. Collet, and O. Alam. Delaying Decisions in Variable Concern Hierarchies. In *ACM SIGPLAN Notices*, volume 52, pages 93–103. ACM, 2016.
29. J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano. Why are Formal Methods not used more Widely? In *Proceedings of the Fourth NASA Formal Methods Workshop*. Citeseer, 1997.
30. 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.
31. G. T. Leavens and Y. Cheon. Design by Contract with JML, Sept. 2006.
32. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual*, May 2013.
33. C. Marché and Y. Moy. The Jessie Plugin for Deductive Verification in Frama-C. *INRIA Saclay Île-de-France and LRI, CNRS UMR*, 2012.
34. Q. McNemar. Note on the Sampling Error of the Difference between Correlated Proportions or Percentages. *Psychometrika*, 12(2):153–157, 1947.
35. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.
36. B. Meyer. Applying Design by Contract. 25(10):40–51, 1992.
37. L. Ochoa, O. González-Rojas, and T. Thüm. Using Decision Rules for Solving Conflicts in Extended Feature Models. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, pages 149–160. ACM, Oct. 2015.
38. 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*, page 2. ACM, 2012.

39. Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking JML Specifications Using an Extensible Software Model Checking Framework. 8(3):280–299, June 2006.
40. J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. In *Safety and Reliability of Software Based Systems*, pages 1–42. Springer, 1997.
41. D. Sannella. *A Survey of Formal Software Development Methods.* University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1988.
42. D. Scheurer, R. Hähnle, and R. Bubel. A General Lattice Model for Merging Symbolic Execution Branches. In *Proceedings of the International Conference on Formal Engineering Methods*, pages 57–73. Springer, 2016.
43. J. M. Schumann. *Automated Theorem Proving in Software Engineering.* Springer Science & Business Media, 2001.
44. N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM, 2015.
45. 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*, 20(3-4):487–517, 2012.
46. T. Thüm, J. Meinicke, F. Benduhn, M. Hentschel, A. von Rhein, and G. Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 177–186. ACM, 2014.
47. T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM, Sept. 2012.
48. T. Thüm, T. Winkelmann, R. Schröter, M. Hentschel, and S. Krüger. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 97–104. ACM, 2016.
49. Why Development Team. Why: A Software Verification Platform. Website. Available online at `http://why.lri.fr/`; visited on December 16th, 2010.
50. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering.* Springer, 2012.