Scalability of Deductive Verification Depends on Method Call Treatment

Alexander Knüppel(✉), Thomas Thüm, Carsten Padylla, and Ina Schaefer

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

Abstract. Today, software verification is vital for safety-critical and security-critical applications applied in industry. However, specifying large-scale software systems for efficient verification still demands high effort and expertise. In deductive verification, design by contract is a widespread software methodology to explicitly specify the behavior of programs using Hoare-style pre- and postconditions in a modular fashion. During verification, a method call can either be replaced by an available method contract or by inlining the method's implementation. We argue that neither approach alone is feasible for verifying real-world software systems. Only relying on method inlining does not scale, as the number of inlined methods may lead to a combinatorial explosion. But specifying software is in itself notoriously hard and time-consuming, making it economically unrealistic to specify large-scale software completely. We discuss circumstances in which one of the two approaches is preferred. We evaluate the program verifier KeY with large programs varying in the number of method calls of each method and the maximum depth of the stack trace. Our analyses show that specifying 10% additional methods in a program can reduce the verification costs by up-to 50%, and, thus, an effective combination of contracting and method inlining is indispensable for the scalability of deductive verification.

Keywords: Deductive verification · Design by contract
Method inlining · Method contracting · KeY · Method call treatment

1 Introduction

A challenging task in software engineering is to reason about the correctness of large programs [32]. *Deductive verification* is a technique focusing on formal program verification by generating proof obligations based on implementation and a formal specification [1,22]. Such proof obligations can then be proved to be correct by interactive and automated program provers [1,26] to ensure that a software system behaves as explicitly specified. In *design by contract*, specifications are typically provided in the form of code annotations [4,17,38]. A developer following this methodology annotates part of the source code, such as methods, with *contracts*. Contracts are inspired by the theory of Hoare triples [31]. That

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 159–175, 2018.
https://doi.org/10.1007/978-3-030-03427-6_15

is, contracts specify *preconditions* that need to be satisfied by callers and *postconditions* that callers can then rely on. Moreover, contracts can have additional information, such as *frame conditions*, which express explicitly what locations (i.e., program variables) a method is allowed to modify [10].

However, deductive verification combined with design by contract is not a widespread methodology in industry. Despite its various advantages, such as an increased trust in the program's correctness, industry sees little benefit in it compared to less demanding approaches (e.g., unit testing). One reason is that it is regareded as cost-ineffective. Indeed, the specification effort is high and error-prone [5] and verification tasks are time-consuming.

In this regard, method call treatment is one critical aspect in the discussion about scability of deductive verification. Generally, there exist three strategies to handle method calls. The first strategy is to always *inline* methods [1]. That is, each method call in the method under verification is replaced by its respective implementation. However, method inlining is often infeasible due to two reasons: (1) it fails per definition in case of recursion or unavailable source code and (2), based on large call stacks, it often results in larger and more complex proof obligations, which are too costly to verify. The second strategy is *method contracting* [35], where we exploit the contract of a called method if available. In this case, the verification consists of checking the preconditions at call-site and abstracting the call by its postcondition. If the implementation of a method specified with a contract changes, and the contract holds after the change, only the method itself had to be verified again. With pure method inlining, always all callers need re-verification. One problem is that not all methods can be specified, as the specification effort is typically too high for large-scale software systems. The third strategy is to use both, method inlining and method contracting, in combination.

Surprisingly, method inlining and method contracting seem to be used interchangeably in research. Numerous evaluations simply use one or the other without justification and often not even let readers *explicitly* know which one was applied (e.g., [1,2]). This is problematic, as research prototypes are typically applied to tiny examples only, where the verification effort is not high enough to see a difference. Hence, when aiming to apply such approaches to large-scale programs in industry, where the specification and verification effort is indeed high, applicability may be unclear.

In this paper, we investigate method call treatment and its scalability in the context of deductive verification empirically to discuss on how to improve the cost-effectiveness of this methodology to become more effective in industrial applications. As a starting point, we employ KeY [1], a static verifier for Java programs with an active community. In KeY, a user either applies pure method inlining or replaces methods with their respective contract (i.e., applying inlining only when no contract is available). To measure the verification effort in a controlled evaluation setting, we generate large, fully specified artificial programs that vary in the number of method calls within each method and the maximum depth of the stack trace. We justify the generation of artificial programs based on three reasons: (1) to the best of our knowledge, no fully-specified real software systems exist that provide a large enough call depth necessary for our evaluation, (2) the specification effort of such systems is too large for us to specify them ourselves, and (3) generating such programs gives us confidence in the correctness of the implementation and specification. In summary, we make the following contributions:

- We discuss method inlining and method contracting for their advantages and limitations in deductive verification and propose to use an effective mixture of both approaches.
- We introduce an artificial benchmark for JML-based verification tools.
- We evaluate our proposal on large generated programs with KeY empirically by measuring the verification effort in different scenarios. Our empirical investigation is a stepping stone towards automated deductive verification and better applicability for industrial use cases.

2 Method Call Treatment in Deductive Verification

Deductive verification is a formal approach to reason about logical properties of programs [43]. Properties such as *"does not crash"*, *"has no arithmetic overflows"*, and even more complex behavioural properties such as *"sorts an array"* are possible. Program verifiers translate these properties and the program to different flavors of first-order logic to reason about their conformance and to become amenable for proof automation [1,4,26]. Deductive program verification is often based on one of two approaches to transform implementation and specification into provable verification conditions, namely *weakest precondition calculus* and *symbolic execution*. Following the original formulation of Floyd-Hoare logic [27,31], Dijkstra suggested the weakest precondition calculus to compute the weakest conditions that must hold at the initial state of the program for given postconditions [22]. Symbolic execution follows a forward manner. Here, all possible execution paths for all possible input data are explored, which can be exploited for program verification against functional properties [15]. Program verification is often not fully automatic due to the undecidability of the underlying verification problem. User interaction, such as providing loop invariants, becomes necessary when proofs exist but cannot be found automatically.

The program verifier KeY allows practitioners to verify that a given Java program (or parts thereof) adheres to its contracts written in the Java Modeling Language (JML). Figure 1 illustrates how JML is used to specify the intended behavior of a Java program in terms of method contracts. Here, method `maxInArray` returns the maximum integer in an integer array and `max` the maximum of two integers. The keyword `requires` represents the precondition that must be fulfilled. The keyword `ensures` represents the postcondition that all callers of that method can rely on when the precondition is fulfilled.

Method `maxInArray` calls method `max` when the size of the array exceeds one. To verify that `maxInArray` indeed conforms to its contract, we must verify that all invoked methods behave correctly as well. Otherwise, we cannot ensure

```
/*@ public normal_behavior
  @ requires array.length >= 1;
  @ ensures \result ==
        (\max int j; 0 <= j && j < array.length; array[j]);
  @*/
public int maxInArray(int array[]){
  int tmp = array[0];
  for(int i = 1; i < array.length; ++i) {
        tmp = max(array[i], tmp);
  }
  return tmp;
}
/*@ public normal_behavior
  @ ensures   a <= b ==> \result == b
  @       && a > b ==> \result == a
  @*/
public int max(int a, int b){
  return a > b ? a : b;
}
```

Fig. 1. Specification of Methods `maxInArray` and `max`

that the method returns the maximum integer in the given array. Typically, two options exist to treat method calls in deductive program verification: *method inlining* and *method contracting*. In this regard, we can either inline the implementation of method `max` into method `maxInArray` at invocation time or we can use the method contract instead.

When verifying a specified method, method contracting only focuses on the first level of method calls, whereas in method inlining the whole call stack of a method is of interest. In our experience, real programs have stack sizes of 20 and beyond. A considerable consequence is thus the increase in size and complexity of the respective proof obligations when method inlining is used, potentially rendering the verification effort infeasible. However, defining strong enough contracts for automated verification is notoriously hard and requires numerous iterations. Even when methods are already specified by developers, a lot of time is spent on refining dependent contracts to make them sufficient for method contracting.

Although examples of real-world software as subject to deductive verification exist in the literature (e.g., TimSort [21] or JavaCard [48]) where a combination of method inlining and contracting was used, the difference in verification effort between both approaches is negligible, because of their comparably small call stacks. To the best of our knowledge, verification effort for either approach of larger programs has not been evaluated empirically before. In the next section, we first discuss under which circumstances method inlining or contracting is preferred.

3 Criteria for Method Call Treatment

Method inlining and method contracting have both their place in deductive verification. In this section, we discuss advantages and drawbacks of both strategies. To this end, we chose criteria important for software engineering and deductive verification involving specification correctness, specification effort, information hiding, unbounded loops, unbounded recursion, verification effort, and incremental verification. In the following discussion, we further distinguish two types of criteria. First, *hard criteria*, where only one approach is applicable in a verification attempt. Second, *soft criteria*, where both approaches are applicable but with varying degrees of success.

3.1 Hard Criteria for Method Call Treatment

Specification Effort. Baumann et al. [5] already argued that writing adequate specifications is the hardest part in formal verification. Typically, developers specify methods concurrently. As a consequence, insufficient specifications or even unspecified methods during the early verification attempts are inevitable. In our experience, writing specifications that are sufficient for method contracting requires numerous iterations over a method itself and its called methods. Moreover, not each developer that specifies contracts is also involved in the verification itself, making it almost impossible to come up with a sufficient method contract for complex behavior from the beginning. Hence, invocations of insufficiently specified methods can only be dealt with through method inlining, when there are no resources available for improving the specification.

Incomplete Implementation. A concern with method inlining when proving correctness of a method is that the implementation of all called methods must be accessible. If source code is unavailable, as often is the case for calls to APIs (e.g., Java's Collection API), or a method is not yet implemented, inlining will fail and correctness cannot be ensured. An exception to the case when source code is unavailable constitutes techniques regarding the verification of bytecode [40]. However, bytecode verification is limited to languages executed by a virtual machine (e.g., Java). Method contracting omits the implementation, as the contract makes the intended behavior of a method explicit. If we assume that all called methods adhere to their respective contract, we can prove correctness with method contracting even when source code is unavailable by providing adequate contracts.

Unbounded Loops and Unbounded Recursion. Dealing with loops and recursion in deductive verification is typically difficult. Bounded loops and bounded recursive calls can be unrolled [1]. When the stop criterion is indefinite at compile-time, loop invariants must be specified and used during verification. For instance, the length of the input array of `maxInArray` is not necessarily known before run-time. If the array's length cannot be determined statically, the `for`-loop becomes unbounded (e.g., the size of the input array is determined at run-time). Inlined methods depending on unbounded loops or recursion may therefore result in a time out and consequently fail. A good choice is to use method contracting instead if a contract is available. In the aforementioned example, when `maxInArray` is called in a method that we want to prove correct, method contracting or an additional loop invariant is the preferred choice to automatically ensure that `maxInArray` does not violate any stated properties. The call could also provide concrete bounds, but would deteriorate software evolution and maintainability.

3.2 Soft Criteria for Method Call Treatment

Verification Effort. Verification effort is typically either measured in terms of *proof steps* of a found proof [1] or in total execution time. Here, we focus on the former measurement, as it is independend of external factors, such as computational power. In particular, reasons for a large number of proof steps are manifold, such as the size and complexity of the program and specification that are subject to verification. For instance, symbolic execution in KeY leads to a step-wise unfolding of Java source code. In case of dynamic dispatch, additional case distinctions have to be made during verification [1]. Unlike method contracting, pure method inlining may lead to a combinatorial explosion in the verification effort, as each method can invoke methods itself that must be also inlined. Nevertheless, there exist cases where sufficient method contracts may also result in larger predicates than the actual implementation. In our evaluation, we aim at empirically investigating the verification effort for method inlining and method contracting in more detail.

Re-verification Effort. When software evolves, each modification may solve prior defects or lead to the introduction of new defects and, thus, re-verification becomes necessary. Because verification is expensive, it is desirable to save verification effort once a program is proved by only re-verifying the parts affected by a change. If the implementation of a specified method changes, only the contract has to be re-established for contracting, whereas for method inlining all callers must be re-verified. If the specification of a method changes, however, only contracting is affected. In this case, the method itself and all of its callers must be re-verified. Moreover, this may involve an adaption of multiple depending contracts. Method inlining is unaffected in this regard, because specifications of inlined methods are ignored. To conclude, we state the hypothesis that the re-verification effort is less with method contracting than with method inlining.

3.3 Summary

Table 1 summarizes our insights of the previous discussion. A "+" means that the respective approach usually performs better and a "–" means that the respective approach fails or performs poorly in that category. The first three rows represent the hard criteria for method call treatment. When dealing with insufficient specifications, only method inlining allows us to verify the correctness of a method.

In case of incomplete implementations and unbounded loops and recursion, however, only method contracting suffices. For the soft criteria, we see verification effort and re-verification effort after a change in the implementation in favor of method contracting. If the specification of a called method changes and method inlining is used, re-verification is not needed. However, if the specification of a proven method changes, method inlining should perform worse depending on call width and call depth of the method. Our goal of the next section is to investigate verification effort under change empirically.

Table 1. Comparison of method contracting and method inlining based on chosen criteria

Criteria	Contracting	Method inlining
Specification effort	–	+
Incomplete implementation	+	–
Unbounded loops and recursion	+	–
Verification effort (initial phase)	+	–
Re-verification effort (change in implementation)	+	–
Re-verification effort (change in specification)	–	+

4 Scalability of Method Call Treatment

We conducted a controlled experiment to measure the effect of method inlining and method contracting on scalability for deductive verification. According to our experience, it is untypical to verify a software system all at once. Hence, we are particularly interested in the effort needed for re-verification of small changes in implementation and specification of single methods. The two independent variables are (a) the *call width* representing the number of method calls within a method body and (b) the *call depth* representing the maximum stack trace of a method call. Both variables allow us to increase the program complexity in a comprehensible and controllable way.

For proving methods in realistic software systems, two problems arise. On the one hand, due to our interest in the verification effort under change, each method needs to be specified with strong enough contracts. The demand in specification effort is, thus, high and not manageable by us for research projects of this scope. On the other hand, for call depths of 20–30, the verification effort becomes infeasible with method inlining. Our solution is therefore to generate artificial programs, varying in call width and call depth, for which we can ensure correct implementations and specifications such that all proofs can be closed automatically.

The rationale behind excluding user interaction is twofold. First, developers need to know how to formally specifiy software, but they should not need to have expertise in proof theory [4]). This, however, is necessary to work with verification systems interactively. Second, the increased expense does not justify the additional insights we may get and is probably unmanageable for us with respect to our experiments.

For the comparison between method inlining and method contracting, we (a) evaluate the root method and a leaf method under change, which gives us indirectly upper and lower bounds for the verification effort of an arbitrary method in-between the call hierarchy, and (b) evaluate the verification effort of ten randomly chosen scenarios, where specifications are sufficient for method contracting in 0%, 10%, ..., 100% of the total number of methods. We address the following research questions.

- **RQ1.1:** What is the re-verification effort if the root method's implementation is changed?
- **RQ1.2:** What is the re-verification effort if the root method's contract is changed?
- **RQ2.1:** What is the re-verification effort if a leaf method's implementation is changed?
- **RQ2.2:** What is the re-verification effort if a leaf method's contract is changed?
- **RQ3:** Given a partially specified program, to what extend does the distribution of contracts impact the verification effort?

Our generators produce Java programs with JML specifications for a given call width and call depth. We analyzed these programs using the program verifier KeY in version 2.6. With the exception to the treatment of method calls, all parameters are set to their respective default. All generators, implementation artifacts, and experimental results can be found online.[1]

4.1 Benchmark for JML-Based Verification

Sorting algorithms are typical in software systems and should ideally be verified to ensure their intended behavior [21]. They are particularly interesting for an evaluation, as they embody real use cases with typical language constructs such as arrays or loops. We therefore decided to implement a sorting algorithm close to bubble sort called *circuitous sorting*, where we are able to specify a variable call width and call depth. However, writing generators with variable call width and call depth to produce verifiable programs with moderate complexity is non-trivial. For the circuitous sorting program we needed countless iterations until all generated method contracts including loop invariants were strong enough for the successful verification with method contracting. Consequently, we decided to write a generator for a simpler program first, namely a program with variable call width and call depth that performs an addition called *circuitous addition*. Our generators together with their respective results may serve as benchmarks

[1] https://www.github.com/AlexanderKnueppel/MethodCallTreatment.

for upcoming techniques that aim at reducing verification effort. In the following, we briefly describe both generators.

Generator for the Circuitous Addition: We built a generator for programs that count and return the number of total method calls plus some input i. The control flow of these programs with call width n and call depth m for the root method `a1` is as follows. First, the method `a1` takes an integer as input and invokes n methods. Each of these methods invokes n methods itself. This procedure goes on until the depth size m is reached. Leaf methods return input i. As depicted in Fig. 2, `a1`'s method contract ensures in its `ensures` statement that i is incremented by 2. In the example, the call depth is set to 1, so `b1` is a leaf method returning input i.

```
/*@ public normal_behavior
@ requires i < 2147483608-2;
@ ensures \result==\old(i)+2;
@*/
public int a1(int i){
  int j = b1(i); i = j+1;
  j = b1(i); i = j+1;
  return i;
}
```

```
/*@ public normal_behavior
@ requires i < 2147483608;
@ ensures \result==\old(i);
@*/
public int b1(int i){
  return i;
}
```

Fig. 2. Root method `a1` and leaf method `b1` of generated circuitous addition program with call width = 2 and call depth = 1.

Generator for Circuitous Sorting: The root method `a1` of the generated circuitous sorting program for call width $n = 2$ is depicted in Fig. 3. Usually, bubble sort is formulated using two nested loops. To integrate call width and call depth, we decomposed the original algorithm into numerous methods accordingly. The leaf method brings exactly one element to its correct sorting position in the input array. The methods on the layer above are calling the leaf method n times and are also bringing one element to the correct position themselves.

Code Optimization. Our generators produce very large programs with many methods for high call widths and call depths. For instance, the add program with call width 9 and call depth 9 allocates approximately 10GB of hard drive. As a result, a huge amount of time is spent in the parsing process of the program in KeY for such programs. We thus simplified our programs such that only one method is created for each layer. The examples in Figs. 2 and 3 are already optimized such that they call method `b1` two times instead of calling a method `b1` and then a method `b2`, both having an equivalent implementation and specification. We checked that the verification effort in KeY is the same for both approaches, which is why we kept this optimization for **RQ1** and **RQ2**. For **RQ3**, we use the former approach, as otherwise we would not be able to have specified and non-specified methods at the same time on the same layer.

```
/*@ public normal_behavior
@ requires p >= 0 && p + 7 < a.length;
@ requires (p > 0 ==>(\forall int y; 0 <= y &&
    y < a.length - p; a[y] <= a[a.length - p]));
@ requires (\forall int z; a.length - p <= z &&
    z < a.length -1; a[z] <= a[z+1]);
@ ensures (\forall int o; 0 <= o &&
    o < a.length - (7+p); a[o] <= a[a.length - (7+p)]);
@ ensures (\forall int pos; a.length - (7+p) <= pos &&
    p < a.length-1; a[p] <= a[p+1]);
@*/
public void a1(int[] a, int p){
  b1(a,p+0);
  b1(a,p+3);
  int b = 0;
/*@ loop_invariant
@ 0 <= b && b < a.length - (p + 6) &&
@ (\forall int k; 0 <= k && k < b; a[k] <= a[b]) &&
@ (\forall int l; a.length - (p + 6) <= l &&
  l < a.length -1; a[l] <= a[l+1]) &&
@ ((p + 6) > 0 ==> (\forall int m; 0 <= m &&
  m < a.length - (p + 6); a[m] <= a[a.length - (p + 6)]));
@ decreasing a.length - b;
@ assignable a[*];
@*/
  while(b < a.length-(1+(p + 6))){
    if(a[b] > a[b+1]){
      int x = a[b];
      a[b] = a[b+1];
      a[b+1] = x;
    }
    b++;
  }
}
```

Fig. 3. Root method `a1` of the circuitous sorting program for call width = 2.

4.2 Empirical Comparison of Method Call Treatment

We now present the results of our study for circuitous addition and circuitous sorting. We decided to use KeY, since a user can choose between the options *method expand* (i.e., method inlining) and method contracting. We conducted all experiments on an infrastructure with two virtual servers, each constituting 16 cores and an assigned RAM of 48 GB. We limited the number of proof steps to 500,000 per experiment, after which we could not observe anymore progress in the verification phase.

RQ1.1: *What is the re-verification effort if the root method's implementation is changed?* If the root method's body is changed, only the root method must be verified again, because the root method has no callers. Figure 4a and b depict the number of proof steps needed to verify the root method of the circuitous addition and circuitous sorting programs for method inlining and method contracting. Y-axes of (a)–(c) have logarithmic scale. The call depth for both programs reaches from 1 to 10. The call width is 5 for the circuitous addition program and 1 for the circuitous sorting program. In our experience, 5 is a realistic number of method calls for good-structured Java programs. However, circuitous sorting was not verifiable anymore for larger call widths. The result were either timeouts or `OutOfMemory`-exceptions. As expected, changing the root method's implementation results in exponential verification effort for method inlining, because every method in the call stack is inlined. For method contracting, we have linear verification effort.

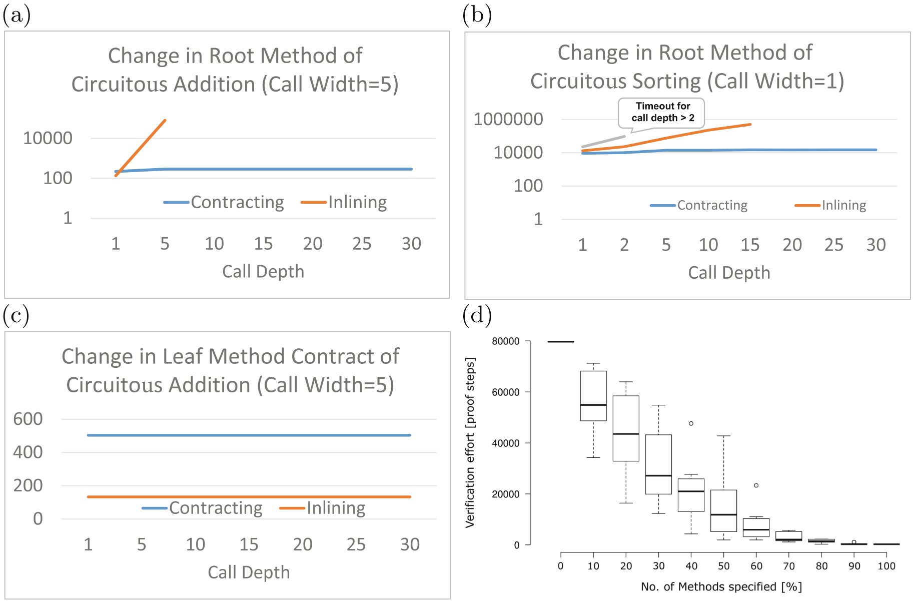

Fig. 4. Empirical results of our evaluation for various change scenarios with respect to call width and call depth.

RQ1.2: *What is the re-verification effort if the root method's specification is changed?* Likewise to RQ1.1, if the root method's specification is changed, only the root method must be verified again. Hence, the result is the same as for *RQ1.1*.

RQ2.1: *What is the re-verification effort if a leaf method's implementation is changed?* For method inlining, every method that inlines the respective leaf method must be verified again. In this case, the verification effort becomes exponential, as every method is inlined at each layer above their own layer exactly once. For method contracting, only the leaf method must be verified and, thus, only a constant number of proof steps is required, independent of the call depth. This case is identical to *RQ2.2* in terms of verification effort.

RQ2.2: *What is the re-verification effort if a leaf method's contract is changed?* If the contract of a leaf method is changed, we have constant verification effort for method inlining and proportional effort to the number of callers for method contracting (cf. Fig. 4c). However, for method inlining we only need to re-verify the leaf method, whereas for method contracting, we need to re-verify the leaf method and every caller of it exactly once, independent of our optimization. The re-verification effort is thus higher with method contracting.

RQ3: *Given a partially specified program, to what extend does the distribution of contracts impact the verification effort?* We evaluated ten scenarios, where the circuitous addition program was specified iteratively. In each iteration, 10% additional randomly-chosen methods are specified. Subsequently, the verification effort of the root method is measured to control call width and call depth. Since KeY does not support a mixture of method contracting and method inlining natively, this way we simulate a combination of both approaches. The box plot in Fig. 4d illustrates the results, where 0% represents method inlining (i.e., only the root method is specified) and 100% represents method contracting (i.e., every method is specified). The analysis emphasizes that strategically specifying an additional 10% of all methods inside the call hierarchy can reduce the verification effort by up-to 50% on average (cf. the median on 20% and 30%). Moreover, there is a wide range in verification effort on numerous iterations. Hence, it matters which parts are specified with respect to the verification effort and, thus, there seems to be potential for a guiding specification process to further support developers.

Threats to Validity. We generated programs ourselves, which threatens internal validity, as we might have chosen unrealistic specifications or implementations. However, the sorting program exhibits that writing generators for specified source code – even for small programs – sufficient for method contracting and automated verification is non-trivial. Moreover, we focused on controlling call width and call depth to investigate the scalability of method call treatment approaches, which otherwise would not have been possible. In this regard, we mostly depicted results for a call width of five, as long methods are considered a bad smell anyway [28]. Moreover, we specified all loops in the sorting program with a loop invariant. Even for method inlining, loop invariants are necessary to find proofs automatically.

An external threat is the choice of KeY as the only verification system. In fact, our evaluation revealed a bug in the implementation leading to an increased garbage collection, which deteriorated the verification time needed. Yet, KeY is one of the most mature verification systems for Java programs with a dedicated community. Moreover, we were not interested in the total number of proof steps needed for either method inlining or method contracting, but how both approaches influence deductive verification in relation. Finally, we generated programs with exactly one class, as we did not want to measure the influence of classes on the verification effort, but only changes in specification and implementation of methods directly.

5 Related Work

Method call treatment is only one parameter of many in the verification of programs. For instance, other parameters in KeY include the treatment of loops (i.e., loop unrolling or using a specified invariant), strategies for proof splitting, and also how arithmetic or quantifiers are treated [1]. However, we were particularly interested in the differences between method inlining and method contracting, which is why we set all other parameters to their defaults.

To give a feeling on how research on deductive verification is considering the treatment of method calls, we inspected numerous publications. In particular, we looked at publications containing empirical evaluations and verification measurements, as well as publications contributing conceptual ideas based upon method contracts. To briefly summarize, we found eight publications that use method contracting [5,16,17,23,24,39,44,49] and seven publications that use method inlining [7,18,20,34,37,42,47]. Numerous other publications we investigated do not give information about the used approach [1,2,6,9,11,13,25,46]. Our evaluation shows that method call treatment is not yet another parameter, but greatly affects the performance of evaluations significantly. Essentially, inlining allows us to trade human time for an increase in machine time. To put claimed evaluation results into perspective, the respective approach to method calls should thus be indicated.

A survey on different languages for behavioral contracts was done by Hatcliff et al. [30]. Besides JML, there are alternatives for specifying Java source code, such as C4J [14] or Contract4J [50]. Other examples of tools for deductive program verification include Dafny [36], VCC [19], Verifast [33], Spec# [4], KIV [41], Why3 [8], and F* [45]. We plan to investigate those tools with respect to method call treatment in the future.

A further abstraction on contracts is provided by means of abstract contracts [29]. Method inlining is prone to changes in the implementation, whereas method contracting is prone to changes in the specification. Abstract contracts delay reasoning about changes of method contracts to the latest stage and, thus, enable sophisticated proof reuse by being less prone to changes.

There exist also alternatives to human-written contracting that were not discussed here. Algorithmic techniques aim at extracting contracts for helper procedures, such as logical abduction [3] or logical interpolation based on Horn clauses [12]. Such techniques can help to achieve efficiency and scalability of verification without significantly increasing the required human specification effort, potentially reducing the latter to human inspection of machine-produced specifications.

6 Conclusion and Future Work

Deductive verification has not found its way into industry yet due to issues with the scalability in specification and verification. We investigated method call treatment, which is an important parameter that needs to be considered when working with deductive verification. Our discussion on the differences between method inlining and method contracting reveals that neither approach is superior in all aspects. Surprisingly, an empirical comparison with respect to the verification effort and specification effort has not been made before. We filled that gap by conducting experiments using the program verifier KeY, in which we used artificial programs varying in the number of method calls of each method and the maximum depth of the stack trace.

A sufficient specification for method contracting demands high effort. The study of the circuitous sorting program showed that even specifying small programs to be sufficient for method contracting is hard. However, mainly relying on method inlining in the verification process leads to scalability problems. In this case, our benchmark revealed that inlining over numerous layers is ineffective for re-verification (i.e., time out for the circuitous addition program over a call depth of five). We thus advocate to use an efficient mixture of both, method inlining and method contracting; putting too much work on less impacting specifications may impair the verification effort significantly, whereas a better prioritization is indispensable when programs are specified and verified incrementally (cf. Fig. 4d).

To make deductive verification scalable, we need to develop strategies for identifying specifications of prime importance that reduce the accumulated verification effort. To investigate how such strategies may look like, it is necessary to verify more fully-specified programs with respect to the number of proof steps needed, and also to evaluate how other parameters and other verifiers influence the verification effort. In particular for KeY, an additional annotation in the source code to indicate method calls that should be inlined (i.e., even in the presence of a contract) could be integrated to allow for a explicit mixture of method inlining and method contracting. To cope with high specification effort in general, stronger tool support is needed for guiding less experienced developers in the specification process.

Acknowledgments. This work was supported by the DFG (German Research Foundation) under the Researcher Unit FOR1800: Controlling Concurrent Change (CCC). We gratefully acknowledge Richard Bubel for fruitful discussions and valuable feedback throughout this work.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification – The KeY Book: From Theory to Practice, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. **77**(12), 1289–1309 (2012)
3. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. ACM SIGPLAN Not. **51**, 789–801 (2016)
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Comm. ACM **54**, 81–91 (2011)

5. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification-specification is the new bottleneck. SSV, pp. 18–32 (2012)
6. Beckert, B., Grebing, S., Böhl, F.: How to put usability into focus: using focus groups to evaluate the usability of interactive theorem provers. In: Workshop on User Interfaces for Theorem Provers (UITP) (2014)
7. Beckert, B., Klebanov, V.: A dynamic logic for deductive verification of concurrent java programs with condition variables. In: Satellite Workshop at CONCUR, p. 3 (2007)
8. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Proceedings of International Workshop on Intermediate Verification Languages, pp. 53–64 (2011)
9. Boldo, S.: Deductive formal verification: how to make your floating-point programs behave. Ph.D. thesis, Université Paris-Sud (2014)
10. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. Softw. Eng. (TSE) **21**(10), 785–798 (1995)
11. Braibant, T., Jourdan, J.-H., Monniaux, D.: Implementing and reasoning about hash-consed data structures in Coq. J. Autom. Reason. **53**(3), 271–304 (2014)
12. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free presburger arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 384–399. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_33
13. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_5
14. Buchwald, H., Meyerer, F.: C4J: Contracts, Java und Eclipse. Eclipse Mag. **13**(3), 64–69 (2013)
15. Burstall, R.: Program Proving as Hand Simulation with a Little Induction. North-Holland, Amsterdam (1974)
16. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Proceedings of International Conference Functional Programming (ICFP), vol. 46, pp. 418–430. ACM (2011)
17. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
18. Cok, D.R., Johnson, S.C.: SPEEDY: an eclipse-based IDE for invariant inference. In: Workshop on Formal Integrated Development Environment (F-IDE), 149 (2014)
19. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: contract-based modular verification of concurrent C. In: Companion International Conference Software Engineering (ICSEC), pp. 429–430. IEEE (2009)
20. de Gouw, S., de Boer, F., Ahrendt, W., Bubel, R.: Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic. Softw. Syst. Model. **15**, 1–24 (2014)
21. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
22. Dijkstra, E.W.: A Discipline of Programming, 1st edn. Prentice Hall PTR, Upper Saddle River (1976)

23. El Ghazi, A.A., Ulbrich, M., Gladisch, C., Tyszberowicz, S., Taghdiri, M.: JKelloy: a proof assistant for relational specifications of java programs. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 173–187. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_13
24. Engel, C.: Deductive verification of safety-critical Java programs. Ph.D. thesis, Karlsruhe Institute of Technology (2009)
25. Filliâtre, J.-C.: Deductive program verification. Ph.D. thesis, Université Paris (2011)
26. Filliâtre, J.-C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
27. Floyd, R.W.: Assigning meanings to programs. Math. Aspects Comput. Sci. **19**, 19–32 (1967)
28. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (2000)
29. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in software verification by abstract method calls. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 300–314. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_21
30. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012)
31. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**(10), 576–580 (1969)
32. Hoare, T.: The verifying compiler: a grand challenge for computing research. In: Böszörményi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789, pp. 25–35. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45213-3_4
33. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
34. Ji, R., Bubel, R.: PE-KeY: a partial evaluator for Java programs. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 283–295. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_20
35. Leavens, G.T., Cheon, Y.: Design by Contract with JML, September 2006
36. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
37. Leino, K.R.M.: Automating induction with an SMT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 315–331. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_21
38. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall Inc., Upper Saddle River (1988)
39. Mostowski, W.: Fully verified Java card API reference implementation. Verify, 7 (2007)
40. Posegga, J., Vogt, H.: Byte code verification for Java smart cards based on model checking. In: Quisquater, J.-J., Deswarte, Y., Meadows, C., Gollmann, D. (eds.) ESORICS 1998. LNCS, vol. 1485, pp. 175–190. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055863

41. Reif, W.: The Kiv-approach to software verification. In: Broy, M., Jähnichen, S. (eds.) KORSO: Methods, Languages, and Tools for the Construction of Correct Software. LNCS, vol. 1009, pp. 339–368. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0015471
42. Schreiner, W.: Computer-assisted program reasoning based on a relational semantics of programs. In: First Workshop on CTP Components for Educational Software (2012)
43. Schumann, J.M.: Automated Theorem Proving in Software Engineering. Springer, Heidelberg (2001). https://doi.org/10.1007/978-3-662-22646-9
44. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. Proc. Symp. Princ. Program. Lang. (POPL) **45**(1), 199–210 (2010)
45. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: Proceedings of Symposium Principles of Programming Languages (POPL), vol. 51, pp. 256–270. ACM (2016)
46. ter Beek, M.H., de Vink, E.P., Willemse, T.A.: Towards a feature mu-Calculus targeting SPL verification. In: Proceedings of International Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE), pp. 61–75 (2016)
47. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: Proceeding of International Conference Generative Programming and Component Engineering (GPCE), vol. 48, pp. 11–20. ACM (2012)
48. Trentelman, K.: Proving correctness of JavaCard DL Taclets using Bali. In: Proceedings of International Conference Software Engineering and Formal Methods (SEFM), pp. 160–169. IEEE (2005)
49. Walter, D.: A formal verification environment for use in the certification of safety-related C-programs. Ph.D. thesis, Bremen, University, Dissertation (2010)
50. Wampler, D.: Contract4J for design by contract in Java: design pattern-like protocols and aspect interfaces. In: Fifth AOSD Workshop on ACP4IS, pp. 27–30. Citeseer (2006)