Program Checking With Less Hassle

Julian Tschannen[1], Carlo A. Furia[1], Martin Nordio[1], and Bertrand Meyer[1,2]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch
[2] ITMO National Research University, St. Petersburg, Russia

Abstract. The simple and often imprecise specifications that programmers may write are a significant limit to a wider application of rigorous program verification techniques. Part of the reason why non-specialists find writing good specification hard is that, when verification fails, they receive little guidance as to what the causes might be, such as implementation errors or inaccurate specifications. To address these limitations, this paper presents *two-step verification*, a technique that combines implicit specifications, inlining, and loop unrolling to provide improved user feedback when verification fails. Two-step verification performs two independent verification attempts for each program element: one using standard modular reasoning, and another one after inlining and unrolling; comparing the outcomes of the two steps suggests which elements should be improved. Two-step verification is implemented in AutoProof, our static verifier for Eiffel programs integrated in EVE (the Eiffel Verification Environment) and available online.

1 The Trouble with Specs

There was a time when formal verification required heroic efforts and was the exclusive domain of a small group of visionaries. That time is now over; but formal techniques still seem a long way from becoming commonplace. If formal verification techniques are to become a standard part of the software development process—and they are—we have to understand and remove the obstacles that still prevent non-specialists from using them.

A crucial issue is specification. Program correctness is a relative notion, in that a program is correct not in absolute terms but only relative to a given *specification* of its expected behavior; in other words, verified programs are only as good as their specification. Unfortunately, many programmers are averse to writing specifications, especially formal ones, for a variety of reasons that mostly boil down a benefit-to-effort ratio perceived as too low. Writing formal specifications may require specialized skills and experience; and the concrete benefits are dubious in environments that value productivity, assessed through simple quantitative measures, more than quality. Why should programmers subject themselves to the taxing exercise of writing specifications in addition to implementations, if there is not much in it for them other than duplicated work?

There are, however, ways to overcome these obstacles. First, not all program verification requires providing a specification because specifications can sometimes be *inferred* from the program text [8,24,18] or from observing common usage patterns [14,37,36]. In particular, some useful specifications are *implicit* in the programming language semantics: types must be compatible; array accesses must be within bound; dereferenced pointers must be non-null; arithmetic operations must not overflow; and so on. Second, programmers are not incorrigibly disinclined to write specifications [7,15,31], provided it does not require subverting their standard programming practices, produces valuable feedback, and brings tangible benefits. Interesting challenges lie in reducing the remaining gap between the user experience most programmer are expecting and the state-of-the-art of formal verification techniques and tools.

In this paper, we combine a series of techniques to improve the applicability and usability of static program checkers such as Spec# [2], Dafny [27], or one of the incarnations of ESC/Java [17,11,20], which verify functional properties of sequential programs specified using contracts (preconditions, postconditions, and class invariants). To enable verification of code with little or no specification, we deploy implicit contracts, routine inlining, and loop unrolling. *Implicit contracts* (described in Section 3) are simple contracts that follow from the application of certain programming constructs; for example, every array access implicitly requires that the index be within bounds. Routine *inlining* (Section 4) replaces calls to routines with the routines' bodies, to obviate the need for a sufficiently expressive callee's specification when reasoning in the caller's context. Similarly, loop *unrolling* makes it possible to reason about loops with incomplete or missing invariants by directly considering the concrete loop bodies.

Implicit contracts, inlining, and unrolling—besides being directly useful to improve reasoning with scarce specification—are the ingredients of *two-step verification* (Section 5), a technique to improve the usability and effectiveness of static checking. Two-step verification performs two verification attempts for every routine: the first one is a standard static checking, using modular reasoning based on programmer-written contracts (whatever they are) plus possibly implicit contracts; the second one uses inlining and unrolling. Comparing the outcomes of the two verification steps provides valuable information about the state of the program and its specification, which is then used to improve the feedback given to users. Bugs violating implicit contracts make for early error detection, and may convince users to add some *explicit* contracts that avoid them. Discrepancies between the verification of calls with and without inlining may help understand whether failed verification attempts are due to errors in the implementation or in the specification, or simply a more accurate specification is required. For example, a call to some routine r that may violate r's user-written precondition but verifies correctly after inlining r's body signals that r's precondition may be unnecessarily strong. Two-step verification is applied completely automatically: users get the most complete feedback based on the integration of the results of the various verification steps—with and without inlining and similar techniques.

We implemented our techniques in AutoProof [34,35], a static verifier for Eiffel programs using Boogie [26] as back-end, and we used it to verify all the examples discussed in the paper. AutoProof is integrated in EVE [33], the research branch of the EiffelStudio development environment, and is freely available online:

`http://se.inf.ethz.ch/research/eve/`

The rest of the paper presents the verification techniques in detail, and illustrates them on non-trivial example programs taken from the VSTTE 2012 verification competition [16] and other software verification benchmarks [1]. Section 6 discusses other programs verified using our approach. We show how the basic techniques and their integration into two-step verification make for better feedback, early error detection, and successful verification even when very limited amounts of specification are available.

2 Overview and Illustrative Examples

Binary search is a widely-known algorithm and is considered a standard benchmark for software verification [1]. Most programmers have implemented it at least once in their life. According to Knuth [23, Vol. 3, Sec. 6.2.1], their implementations were often "wrong the first few times they tried".

```
1  binary_search (a: ARRAY [INTEGER]; x: INTEGER): INTEGER
2    require a ≠Void
3    local middle: INTEGER
4    do
5      if a.count = 0 then
6        Result := −1
7      else
8        middle := (1 + a.count) / 2
9        if a[middle] = x then
10         Result := middle
11       elseif a[middle] > x then
12         Result := binary_search (a[1:middle − 1], x)
13       else
14         assert a[middle] < x end
15         Result := binary_search (a[middle + 1:a.count]), x)
16         if Result ≠−1 then Result := Result + middle end
17       end
18     end
19   ensure Result = −1 or (1 ≤Result and Result ≤a.count)
```

Fig. 1. An implementation of binary search.

To demonstrate, consider the binary search implementation in Figure 1,[3] which takes an integer array a and an integer value x and returns an integer index (the value assigned to **Result**) pointing to an occurrence of x in a. Since we assume arrays numbered from one, if x is not found in a the routine by convention returns -1. The implementation in Figure 1 is indicative of what programmers typically write[30,15] when using a language supporting specifications in the form of contracts (pre- and postconditions, and intermediate assertions such as loop invariants and **assert** instructions): the implementation is "almost" correct (if you do not immediately see the error, read on), and the specification is obviously incomplete.

Part of the missing specification is implicit in the semantics of the programming language, which is probably why the programmer did not bother writing it down explicitly. In particular, arithmetic operations should not overflow for the program to have a well-defined semantics. The midpoint calculation on line 8 overflows when the array size $a.count$ has value equal to the largest representable machine integer, even if the value of *middle* is within the bounds; this is indeed a common error in real implementations of binary search [4].

If we try to verify the program in Figure 1 using static verifiers such as Dafny, we do not find any error because integer variables are modeled using mathematical integers which do not overflow. In Section 3 we discuss our approach which automatically instantiates implicit specification elements that represent tacit assumptions about the programming language semantics. Such *implicit contracts* help early error detection of subtle errors not explicitly specified, such as the potential overflow just discussed. In addition, they are made available within a more general static verification mechanism, where they can complement programmer-written contracts to improve the efficiency of the overall verification process without sacrificing precision.

Not only do incomplete specifications limit the kinds of error that can be detected automatically during verification; they may also prevent verifying perfectly *correct* programs as we now illustrate with the example of Figure 2, taken from the VSTTE 2012 verification competition. Routine *two_way_sort* sorts an array a of Boolean values in linear time with a technique similar to the partitioning algorithm used in Quick Sort. Two pointers i and j scan the array from its opposite ends; whenever they point to an inversion (that is, a **False** in the right-hand side and a **True** in the left-hand side) they remove it by swapping the elements pointed. When the whole array is scanned, it is sorted.

The sorting algorithm calls an auxiliary routine *swap* that exchanges elements; *swap* does not have any specification—again, a situation representative of how programmers typically specify their programs. This is a problem because static verification uses specifications to reason *modularly* about routine calls: the effects of the call to *swap* on line 14 are limited to *swap*'s postcondition. Since it does not have any, the proof of *two_way_sort* does not go through; in particular, it cannot establish that the loop invariant at line 7 is inductive, which would

[3] All code examples are in Eiffel, with few occasional notational simplifications; even readers not familiar with the language should find the code easy to understand.

```
1   two_way_sort (a: ARRAY [BOOLEAN])
2     require a.count >0
3     local i, j: INTEGER
4     do
5       i := 1 ; j := a.count
6       until i ≥ j
7       invariant 1 ≤ i and i ≤ j + 1 and j ≤ a.count
8       loop
9         if not a[i] then
10          i := i + 1
11        elseif a[j] then
12          j := j − 1
13        else
14          swap (a, i, j)
15          i := i + 1
16          j := j − 1
17        end
18      variant j − i + 1 end
19    end
20
21  swap (b: ARRAY [BOOLEAN]; x, y: INTEGER)
22    local t: BOOLEAN
23    do t := b[x] ; b[x] := b[y] ; b[y] := t end
```

Fig. 2. An implementation of two-way sort of Boolean arrays.

then be the basis to establish the variant as well as any programmer-written postcondition.

In our approach, when modular verification fails the verifier makes another attempt after *inlining* routine bodies at their call sites. As we describe in Section 4, the application uses simple heuristics to avoid combinatorial explosion (for example, in the case of recursive calls). With inlining, we can prove that the invariant at line 7 is inductive without need for more specification.

Using inlining, we can also check interesting properties about *clients* of *two_way_sort*. For example, when calling the routine on an empty array, we compare a failed modular verification attempt (which reports a violation of *two_way_sort*'s precondition) to a successful verification with inlining (which only evaluates the routine's body); the discrepancy *suggests* that *two_way_sort*'s precondition is unnecessarily strong and can be relaxed to $a \neq$ **Void** without affecting the rest of the verification process. This kind of improved feedback, concocted from two different verification attempts, is the two-step verification we present in detail in Section 5.

3 Implicit Contracts

The first ingredients of our two-step verification approach are *implicit contracts*: simple specification elements that are implicit in the semantics of the programming language—Eiffel in our examples. Since they are implicit, programmers tend to reason informally about the program without writing them down as assertions. This limits the kinds of properties that can be proved automatically with a static verifier. With implicit contracts, the verifier transparently annotates the program under verification so that the feedback to users is more accurate and goes deeper than what would have been possible based on the explicitly written contract only. We currently support the following classes of implicit contracts.

3.1 Targets Non-Void

A qualified call $t.r_1(a_1).\ldots.r_n(a_n)$, with $n \geq 0$, is non-**Void** if $t \neq$ **Void** and, for $1 \leq k < n$, $t.r_1(a_1).\ldots.r_k(a_k)$ returns a non-**Void** reference. For every such qualified call appearing as instructions or in expressions, we introduce the corresponding implicit contract that asserts that the call is non-**Void**.

For example, *two_way_sort*'s precondition (Figure 2) is augmented with the implicit contract that $a \neq$ **Void** following from the qualified call $a.count$.

3.2 Routine Calls in Contracts

In programming languages supporting contracts there need not be a sharp distinction between functions used in the implementation and functions used in the specification. Routine *two_way_sort*, for example, uses the function call $a.count$—returning the length of array a—in its precondition and loop invariant, but also in the assignment instruction on line 5. Functions used in contracts may have preconditions too; programmers should make them explicit by replicating them whenever the function is mentioned, but they often neglect doing so because it is something that is implicit when those functions are used in normal instructions, whereas it is not checked when the same functions are used in contracts.

Consider, for instance, a function *is_sorted* with the obvious semantics, and suppose that its precondition requires that it is applied to non-empty lists. If *is_sorted* is called anywhere in the implementation, then it is the caller's responsibility to establish its precondition; the caller is aware of the obligation explicit in *is_sorted*'s contract. But if *is_sorted* is called, say, as precondition of *binary_search*, establishing *is_sorted*'s precondition is now the responsibility of callers to *binary_search*, who are, however, unaware of the non-emptiness requirement *implicit* in *binary_search*'s precondition. In fact, the requirement should explicitly feature as one of *binary_search*'s preconditions.

To handle such scenarios automatically, for every call to any function f appearing in contracts, we introduce the corresponding implicit contract that asserts that f's precondition holds right before f is evaluated in the contract. If f's precondition includes calls to other functions, we follow the transitive closure of the preconditions, also checking well-formedness (that is, no circularity occurs).

3.3 Array Accesses

For every array access of the form $a[exp]$ appearing in instructions or in expressions, we introduce the implicit contract $a.lower \leq exp$ **and** $exp \leq a.upper$ which asserts that the expression used as index is within the array's bounds.

In *binary_search*, for example, the accesses $a[middle]$ determine the implicit contract that *middle* is between 1 and $a.count$.

3.4 Arithmetic Expressions

The subexpressions $\mathsf{sub}(e)$ of an integer expression e are defined in the obvious way: if e is an integer constant or an integer variable then $\mathsf{sub}(e) = \{e\}$; if e is the application of a unary operator $\sim$, that is $e = \sim d$, then $\mathsf{sub}(e) = \{e\} \cup \mathsf{sub}(d)$; if e is the application of a binary operator $\oplus$, that is $e = c \oplus d$, then $\mathsf{sub}(e) = \{e\} \cup \mathsf{sub}(c) \cup \mathsf{sub}(d)$. For every integer expression e appearing in instructions or expressions, we introduce the implicit contract that asserts that no subexpression of e's may overflow:

$$\bigwedge_{x \in \mathsf{sub}(e)} \{INTEGER\}.min\_value \leq x \quad \textbf{and} \quad x \leq \{INTEGER\}.max\_value$$

For every subexpression of the form $c \oslash d$, where $\oslash$ is some form of integer division, we also introduce the implicit contract $d \neq 0$, which forbids division by zero.

The integer expression at line 8 in Figure 1, for example, determines the implicit contract $1 + a.count \leq \{INTEGER\}.max\_value$, which may not hold.

4 Inlining and Unrolling

Inlining and unrolling are routinely used by compilers to optimize the generated code for speed; they are also occasionally used for program checking, as we discuss in Section 7. The novelty of our approach is the automatic combination, in two-step verification, of inlining and unrolling with modular "specification-based" verification. Inlining and unrolling may succeed in situations where little programmer-written specification is available; in such cases, users get a summary feedback that combines the output of each individual technique and is aware of the potential unsoundness of inlining and unrolling. The combined feedback gives specific suggestions as to what should be improved. This section presents the definitions of inlining and unrolling; Section 5 discusses how they are combined in two-step verification.

4.1 Inlining

The standard approach to reasoning about routine calls is *modular* based on specifications: the effects of a call to some routine r within the callee are postulated to coincide with whatever r's specification is. More precisely, the callee

should establish that r's precondition holds in the calling context; and can consequently assume that r's postcondition holds and that the call does not modify anything outside of r's declared frame.

The modular approach is necessary to scale verification to large pieces of code. At the same time, it places a considerable burden on programmers, since every shortcoming in the specifications they provide may seriously hinder what can be proved about their programs. This is a practical issue especially for helper functions that are not part of the public API: programmers may not feel compelled to provide accurate specifications—postconditions, in particular—for them because they need not be documented to clients; but they would still like to benefit from automated program checking. This is the case of routine *swap* in Figure 2, which is not specified but whose semantics is obvious to every competent programmer.

Inlining can help in these situations by replacing abstract reasoning based on specifications with concrete reasoning based on implementations whenever the former are insufficient or unsatisfactory. In particular, inlining is likely to be useful whenever the inlined routine has no postcondition, and hence its effects within the callee are undefined under modular reasoning. Of course, inlining has scalability limits; that is why we apply it in limited contexts and combine it with standard modular verification as we discuss in Section 5.

Definition of inlining. Consider a routine r of class C with arguments a, which we represent as:

$$r\ (t\colon\ C\ ;\ a)\ \textbf{require}\ P_r\ \textbf{modify}\ F_r\ \textbf{do}\ B_r\ \textbf{ensure}\ Q_r\ \textbf{end}$$

For $n \geq 0$, the n-inlining $\mathsf{inline}(A, n)$ of calls to r in a piece of code A is defined as A with every call $u.r(b)$ on target u (possibly **Current**) with actual arguments b modified as follows:

$$\mathsf{inline}(u.r(b), n) = \begin{cases} \textbf{assert}\ P_r[u, b]\ ;\ \textbf{havoc}\ F_r[u, b]\ ;\ \textbf{assume}\ Q_r[u, b] & \text{if } n = 0 \\ \mathsf{inline}(B_r[u, b], n - 1) & \text{if } n > 0 \end{cases}$$

Inlining works recursively on calls to routines other than r and recursive calls to r in B_r; non-call instructions are instead unchanged. 0-inlining coincides with the usual modular semantics of calls based on specifications. Otherwise, inlining replaces calls to r with $B_r[u, b]$ (r's body applied to the actual target u and arguments b of the calls), recursively for as many times as the recursion depth n.

Since inlining discards the inlined routine's precondition, it may produce under- or over-approximations of the calls under modular semantics, respectively if the declared precondition is weaker or stronger than the body's weakest precondition.

For any $n > 0$, the n-inlining of *swap* in *two_way_sort*'s body (Figure 2) consists of replacing the call to *swap* at line 14 with *swap*'s body instantiated in the correct context, that is $t := a[i]\ ;\ a[i] := a[j]\ ;\ a[j] := t$ with t a fresh local variable declared inside *two_way_sort*.

Inlining and dynamic dispatching. In programming languages with dynamic dispatching, the binding of routine bodies to routine calls occurs at run-

time, based on the dynamic type of the call targets. This is not a problem for modular reasoning because it can rely on behavioral subtyping and the rule that routine redefinitions in descendants (overriding) may only weaken preconditions and strengthen postconditions. Inlining, instead, has to deal with dynamic dispatching explicitly: in general, verification using inlining of a routine r of class C is repeated for every overriding of r in C's descendants. This also requires to re-verify the system whenever new descendants of C are added, unless overriding r is eventually forbidden (**frozen** in Eiffel, **final** or **private** in Java, or **sealed** in C#). These limitations are, however, not a problem in practice when we apply inlining not indiscriminately but only in limited contexts for small helper routines, and we combine its results with classic modular reasoning as we do in two-step verification.

4.2 Unrolling

The standard approach to modular reasoning also applies to loops based on their loop invariants: the effects of executing a loop on the state of the program after it are postulated to coincide with the loop invariant. The inductiveness of the invariant is established separately for a generic iteration of the loop, and so is the requirement that the invariant hold upon loop entry.

This reliance on expressive loop invariants is at odds with the aversion programmer typically have at writing them. This is not only a matter of habits, but also derives from the fact that loop invariants are often complex specification elements compared to pre- and postconditions [18]; and, unlike pre- and postconditions which constitute useful documentation for clients of the routine, loop invariants are considered merely a means to the end of proving a program correct. The loop of *two_way_sort* in Figure 2, for example, has a simple loop invariant that only bounds the values of the indexes i and j; this prevents proving any complex postcondition.

Unrolling can help in these situations by evaluating the effects of a loop in terms of its concrete body rather than its invariant. This may help prove the postcondition when the invariant is too weak, showing that a certain number of repetitions of the body are sufficient to establish the postcondition. Furthermore, in the cases where we have a way to establish a bound on the number of loop iterations, unrolling precisely renders the implementation semantics. We will generalize these observations when discussing how unrolling is applied automatically in the context of two-step verification (Section 5).

Definition of unrolling. Consider a generic annotated loop L:

$$\textbf{until } \mathit{exit} \textbf{ invariant } I \textbf{ loop } B \textbf{ variant } V \textbf{ end}$$

which repeats the body B until the exit condition *exit* holds, and is annotated with invariant I and variant V. For $n \geq 0$, the n-unrolling $\mathsf{unroll}(L, n)$ of L is defined as:

$$\mathsf{unroll}(L, n) \;=\; (\textbf{if not } \mathit{exit} \textbf{ then } B \textbf{ end})^n$$

where the nth exponent denotes n repetitions. Since unrolling ignores the loop invariant, it may produce under- or over-approximations of the loop's modular

semantics, respectively if the declared loop invariant is weaker or stronger than the body's weakest precondition.

5 Two-step Verification

We have introduced all the elements used in two-step verification; we can finally show how they are combined to produce improved user feedback.

Implicit contracts are simply added whenever appropriate and used to have early detection of errors violating them. In AutoProof, which translates Eiffel to Boogie to perform static proofs, implicit contracts are not added to the Eiffel code but are silently injected into the Boogie translation, so that the input code does not become polluted by many small assertions; users familiar with Eiffel's semantics are aware of them without explicitly writing them down. Errors consisting of violations to implicit contracts reference back the original statements in Eiffel code from which they originated, so that the error report is understandable without looking at the Boogie translation.

Whenever the verifier checks a routine that contains routine calls, two-step verification applies inlining as described in Section 5.1. Whenever it checks a routine that contains loops, two-step verification applies unrolling as described in Section 5.2. The application of the two steps is completely automatic, and is combined for routines that includes both calls and loops; users only get a final improved error report in the form of suggestions that narrow down the possible causes of failed verification more precisely than in standard approaches. Section 5.4 briefly illustrates two-step verification on the running example of Section 2.

5.1 With Inlining

Consider a generic routine r with precondition P_r and postcondition Q_r, whose body B_r contains a call $t.s(a)$ to another routine s with precondition P_s, postcondition Q_s and body B_s (as shown in Figure 3). Two-step verification runs two verification attempts on r:

1. **Modular verification:** The first step of two-step verification for r follows the standard modular verification approach: it tries to verify that r is correct with respect to its specification, using s's specification only to reason about the call to s; and then it separately tries to verify s against its own specification.
2. **Inlined verification:** The second step of two-step verification for r replaces the call to s in r with $\mathsf{inline}(t.s(a), n)$, for some $n > 0$ picked as explained in Section 5.3, and then verifies r with this inlining.

Each of the two steps may fail or succeed. According to the combined outcome, we report a different suggestion to users, as summarized in Table 1.

Precondition fails. If modular verification (first step) fails to establish that s's precondition P_s holds right before the call, but both modular verification of

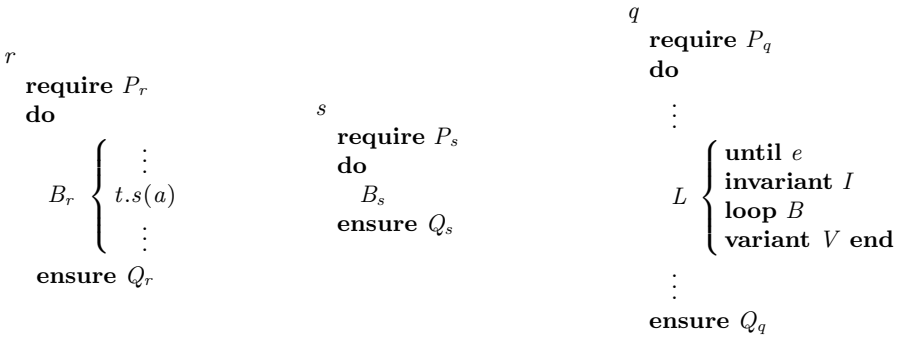

Fig. 3. A routine r calling another routine s; and a routine q with a loop.

step 1: modular		**step 2: inlined**	
VERIFY r	VERIFY s	VERIFY r	SUGGESTION
P_s fails	success	success	weaken P_s or use inlined s
Q_r fails	success	success	strengthen Q_s or use inlined s
success	Q_s fails	success	strengthen P_s or weaken Q_s

Table 1. Two-step verification with inlining: summary of suggestions.

s and inlined verification (second step) of r succeed, it means that s's precondition may be inadequate[4] while its implementation is correct with respect to its specification and to the usage made within r. In this case, there are two options: if s is a helper function used only in r or anyway in limited contexts, we may as well drop s's specification and just use it inlined wherever needed during verification. In more general cases, we should try to *weaken* P_s in a way that better characterizes the actual requirements of how s is used.

Postcondition fails. If modular verification fails to establish that r's postcondition Q_r holds when B_r terminates, but both modular verification of s and inlined verification of r succeed, it means that s's postcondition fails to characterize r's requirements while s's implementation is correct with respect to its specification. As in the previous case, there are two options: we may drop s's specification and just use it inlined; or we should try to *strengthen* Q_s in a way that better characterizes the actual expectations of r on s. A similar scheme applies not just to failed postconditions Q_r but whenever modular verification fails to verify intermediate assertions occurring on paths after the call to s in r.

Local proof fails. If modular verification fails to establish that s's postcondition Q_s holds when B_s terminates, but both modular verification of r and inlined verification of r succeed, it means that s's specification cannot be proved

[4] As with all failed static verification attempts, we cannot exclude that failure is simply due to limitations of the theorem prover.

consistent with its implementation, while the latter is correct with respect to the usage made within r. The suggestion is to change s specification in a way that still accommodates its usage within r and can be verified: strengthen the precondition P_s, weaken the postcondition Q_s, or both. With this information, there is no way to decide if the problem is with the pre- or postcondition, but we can always try to modify either one and run verification again to see if the suggestion changes.

In the remaining cases, two-step verification is inconclusive in the sense that it gives the same feedback as modular verification alone. In particular, when the second step fails it is of no help to determine whether the problem is in the specification or the implementation. If, for example, both modular and inlined verification of r fail to establish the postcondition Q_r, but modular verification of s succeeds, we cannot conclude that s's implementation is wrong because it does not achieve Q_r: it may as well be that r's implementation is wrong, or r's postcondition is unreasonable; which is exactly the information carried by a failed modular verification attempt.

Also notice that inlined verification cannot fail when modular verification fully succeeds: if s's implementation satisfies its specification, and that specification is sufficient to prove r correct, then the semantics of s within r is sufficient to prove the latter correct. Therefore, we need not run the second step when the first one is successful.[5]

5.2 With Unrolling

Consider a generic routine q with precondition P_q and postcondition Q_q, whose body B_q contains a loop L with exit condition e, invariant I, variant V, and body B (as shown in Figure 3). Two-step verification runs two verification attempts on r:

1. **Modular verification:** The first step of two-step verification for r follows the standard modular verification approach: it tries to verify that r is correct with respect to its specification, using the loop invariant I only to reason about the effect of L within r; and then it separately tries to verify that I is a correct inductive loop invariant (that is, it holds on entry and is maintained by iterations of the loop).
2. **Unrolled verification:** The second step of two-step verification for r replaces the loop L in r with $\mathsf{unroll}(L, n)$, for some $n > 0$ picked as explained in Section 5.3, and then verifies r with this unrolling and the additional assertion **assert** $V \leq n$, evaluated upon loop entry, that the loop executes at most n times.[6]

[5] Again, exceptions might occur due to shortcomings of the theorem prover used by the modular verifier, which might be able to prove a set of verification conditions but fail on a syntactically different but semantically equivalent set due to heuristics or limitations of the implementation. These are, however, orthogonal concerns.

[6] If a variant is not available or cannot be verified to be a valid variant, we proceed as if the assertion did not hold.

step 1: modular	**step 2: unrolled**		
VERIFY r	**assert** $V \leq n$	VERIFY r	SUGGESTION
Q_q fails	success	success	use inlined L
Q_q fails	fail	success	strengthen I to generalize
I fails inductiveness	success	success	use inlined L
I fails inductiveness	fail	success	change I to generalize

Table 2. Two-step verification with unrolling: summary of suggestions.

Each of the two steps may fail or succeed. According to the combined outcome, we report a different suggestion to users, as summarized in Table 2.

Postcondition fails. Suppose that modular verification (first step) fails to establish that r's postcondition Q_q holds when B_q terminates, but unrolled verification (second step) of r succeeds. The suggestion in this case depends on whether the prover can also establish the intermediate assertion **assert** $V \leq n$. If it does, n is a finite upper bound on the number of loop iterations in every execution. Thus, the loop implementation is correct but the loop invariant I is inadequate to prove the postcondition; we may as well drop the invariant I and just use the exhaustively unrolled loop during verification. In the more general case where the assertion $V \leq n$ fails, the successful unrolled proof shows that the loop body works with a finite number of iterations, and hence it is likely correct; we may then try to strengthen (or otherwise change) the invariant I in a way that captures a generic number of loop iterations and is sufficiently strong to establish Q_q. A similar scheme applies not just to failed postconditions Q_q but whenever modular verification fails to verify intermediate assertions occurring on paths after the loop L in q.

Invariant fails. Suppose that modular verification fails to establish that I is inductive, but unrolled verification of r succeeds. The suggestion depends on whether the prover can also establish the intermediate assertion **assert** $V \leq n$. If it does, the loop implementation is correct but the loop invariant I is inadequate; we may as well drop the invariant I and just use the exhaustively unrolled loop during verification. In the more general case where the assertion $V \leq n$ fails, the successful unrolled proof shows that the loop body works with a finite number of iterations, and hence it is likely correct; we may then try to change the invariant I in a way that captures a generic number of loop iterations and is sufficiently strong to establish Q_q. With this information, there is no way to decide if the invariant should be strengthened or weakened, but we can always try either one and run verification again.

In the remaining cases, two-step verification gives the same feedback as modular verification alone. And, as for inlining, we need not run the second step (unrolled verification) when modular verification is completely successful.

5.3 Bounds for Nesting and Loops

The application of inlining and unrolling requires a parameter n: the maximum depth of nested calls in the former case; and the number of explicit iterations

of the loop in the latter. The choice of n is more subtle for unrolling—where it should represent a number of iterations sufficient to make the second step of verification succeed—than for inlining—where it becomes relevant only in the presence of nested calls or recursion.

In our implementation, we use simple heuristics to pick values for n that are "feasible", that is do not incur combinatorial explosion. In the case of inlining, we get a crude estimate of the size of the inlined program as follows. For a routine p, let π_p denote the total number of simple paths in p from entry to exit. If p has size $|p|$ (measured in number of instructions) and includes m calls to routines $r_1, \ldots, r_m$, we recursively define $\|p\|_n$ as: $|p|$ if $n = 0$, $|p| + \pi_p(\|r_1\|_{n-1} + \cdots + \|r_m\|_{n-1})$ if $n > 0$. Inlining in p uses an n such that $\|p\|_n \leq 10^4$.

In the case of unrolling a loop within routine q, our implementation does some simple static analysis to determine if the calling context of q or q's precondition suggest a finite bound of the loop (in practice, this is restricted to loops over arrays that are declared statically or with a constant upper bound in the precondition). In such cases, n is simply the inferred bound. Otherwise, we roughly estimate the size of an unrolled loop L as $n|L|$, where $|L|$ is the size of L in number of instructions; unrolling L uses an n such that $n|L| \leq 10^3$.

In many practical cases (in the absence of recursion or deeply nested calls), very small n's (such as $1 \leq n \leq 5$) are sufficient to produce a meaningful results in two-step verification.

5.4 Examples

Let us demonstrate how two-step verification works on the running examples introduced in Section 2. Figure 4 shows two clients of routine *two_way_sort* (Figure 2). Routine *client_1* calls *two_way_sort* on an empty array, which is forbidden by its precondition. Normally, this would be blamed on *client_1*; with two-step verification, however, the second verification attempt inlines *two_way_sort* within *client_1* and successfully verifies it. This suggests that *client_1* is not to blame, because *two_way_sort*'s precondition is unnecessarily strong (first case in Table 1), which is exactly what AutoProof will suggest in this case as shown in Figure 5. In fact, the sorting implementation also works on empty arrays, where it simply does not do anything.

```
1   client_1
2     local a: ARRAY [BOOLEAN]
3     do
4       a := [ ] -- empty array
5       two_way_sort (a)
6     end
```

```
7   client_2
8     local a: ARRAY [BOOLEAN]
9     do
10      a := [True, False, False, True]
11      two_way_sort (a)
12      assert a[1] = False
13    end
```

Fig. 4. Clients of *two_way_sort*.

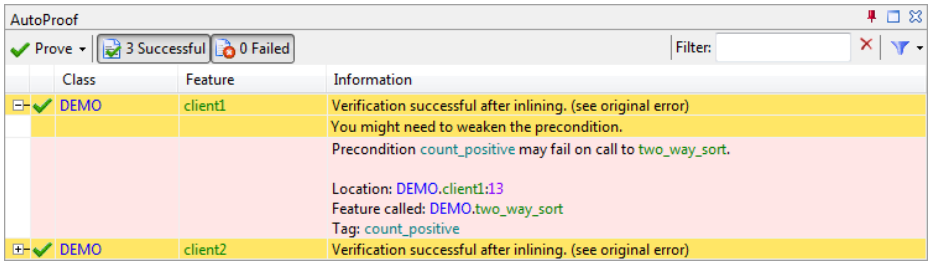

Fig. 5. AutoProof showing feedback of two-step verification.

Routine *client_2* calls *two_way_sort* on a four-element array and checks that its first element is **False** after the call. Modular verification cannot prove this assertion: *two_way_sort* has no postcondition, and hence its effects within *client_2* are undefined. The second verification attempt inlines *two_way_sort* and unrolls the loop four times (since it notices that the call is on a four-element array); this proves that the first array element is **False** after the call (first line in Table 2). In all, *two_way_sort* is not to blame because its implementation works correctly for *client_2*. As summarized in the second line of Table 1, the user can either just be happy with the result or endeavor to write down a suitable postcondition for *two_way_sort* so that the correctness proof can be generalized to arrays of arbitrary length.

Suppose we provide a postcondition that specifies sortedness: using Eiffel's syntax, **across** $1..(a.count-1)$ **as** k **all** $(a[k] = a[k+1])$ **or** $(a[k] \neq a[k+1]$ **and** $a[k] =$ **False**). Modular verification of *two_way_sort* fails to prove this postcondition because the loop invariant at line 7 does not say anything about the array content. Two-step verification makes a second attempt where it unrolls the loop a finite number of times, say 5, and inlines *swap*. The situation is in the second entry of Table 2: we cannot verify that the arbitrary bound of five iterations generally holds (that is $j - i + 1 \leq 5$ holds before the loop), but the success of unrolling in this limited case suggests that *two_way_sort*'s implementation is correct. If we want to get to a general proof, we should improve the loop invariant, and this is precisely the suggestion that two-step verification brings forward.

6 Evaluation

The examples of the previous sections have demonstrated the kind of feedback two-step verification provides. This section contains a preliminary evaluation of the scalability of two-step verification.

Table 3 lists the example programs. The first labeled column after the program name contains the size of the implementation (not counting specification elements) in lines of code. The rest of the table is split in two parts: the first one contains data about two-step verification; the second one the same data about modular verification. The data reported includes: the amount of specification

		TWO-STEP				MODULAR			
		P Q I A				*P Q I A*			
	code	*spec*		*Boogie*	*time*	*spec*		*Boogie*	*time*
1. Maximum	32	1 3 0 0	4	1541	2.06	1 3 4 0	8	712	1.05
2. Sum and Max	32	3 1 0 0	4	1619	2.18	3 1 5 0	9	720	1.04
3. Two-way Sort	44	2 1 0 0	3	1803	2.35	2 1 5 0	8	742	1.06
4. Dutch Flag	45	2 1 0 0	3	1955	2.94	2 1 10 0	13	786	1.14
5. Longest common prefix	30	3 4 0 0	7	1585	2.10	3 4 7 0	14	730	1.05
6. Priority queue	119	0 0 0 7	7	2896	3.35	5 13 5 7	30	1088	1.56
7. Deque	127	0 0 0 11	11	1856	2.51	7 18 4 11	40	1230	1.59
8. Binary Search	48	0 0 0 1	1	2479	3.21	2 3 0 1	6	672	0.99
Total	*477*	*11 10 0 19*	*40*	*15734*	*20.70*	*25 44 40 19*	*128*	*6680*	*9.48*

Table 3. Comparison of two-step and modular verification on selected examples.

necessary to successfully verify the example (number of annotations, split into preconditions P, postconditions Q, invariants I (loop invariants in examples 1–5; class invariants in examples 6–8), and intermediate assertions A); the size (in lines) of the Boogie code generated by AutoProof; and the time in seconds. Two-step verification includes modular verification as first step, but normally requires less specification to be successful; correspondingly, the Boogie code and the time in the first part of the table sum up both steps.

The examples include: (1) finding the maximum in an array, from the COST 2011 verification competition [6]; (2) computing maximum and sum of the elements in an array, from the VSTTE 2010 verification competition [22]; (3) the two-way sort algorithm of Section 2, from the VSTTE 2012 verification competition [16]; (4) Dikstra's Dutch national flag algorithm [13]; (5) computing the longest common prefix of two sequences, from the FM 2012 verification competition [19]; (6) a priority queue implementation, from Tinelli's verification course [32]; (7) a double-ended queue [23, Vol. 1, Sec. 2.2.1]; and (8) the binary search algorithm of Section 2, from the software verification benchmarks [1].

In the experiments with the algorithms 1–5, two-step verification succeeds with loop unrolling of depth $n = 6$, which corresponds to input arrays of the same length. The outcome suggests either to use the unrolled loop, for inputs of bounded length; or to write a suitable loop invariant. A correct loop invariant is necessary for modular verification alone to succeed, in which case the proof generalizes to arrays of arbitrary length. We prove the following postconditions: (1) the output is the array maximum; (2) the output sum is less than or equal to the output maximum times the array length; (3) sortedness of the output, as

formalized in Section 5.4; (4) the output is partitioned in the three flag colors; (5) the output is the longest common prefix of the input array.

In the experiments 6–8 we verify *clients* of the queue, double-ended queue, and binary search, which call some routines and then formalize their expectations on the result with **assert**s after the call. The called routines have no specification (in particular, no postcondition); two-step verification verifies the clients using inlining of the callee, and suggests to add postconditions to generalize the proofs. The postconditions are necessary for modular verification alone to succeed.

The evaluation suggests that two-step verification can check the implementation even when little or no specification is given; its feedback may then help write the necessary specifications to generalize proofs for modular verification.

The runtime overhead of performing two verification steps instead of one is roughly linear in all examples; in fact, unrolling and inlining blow up mainly in the presence of recursion. To better assess how they scale, we have repeated two-step verification of examples 3 (using unrolling) and 6 (using inlining in the presence of recursion) for increasing value of the bound n. Table 4 shows the results in terms of size of the generated Boogie code (in lines) and time necessary to verify it (in seconds). Unrolling scales gracefully until about $n = 10$; afterward, the time taken by Boogie to verify increases very quickly, even if the size of the Boogie code does not blow up. Inlining is more sensitive to the bound, since the size of the inlined code grows exponentially due to the conditional branch in *binary_search*'s body; the time is acceptable until about $n = 7$. Notice that the heuristics for the choice of n discussed in Section 5.3 would generate running times in the order of tens of seconds, thus enforcing a reasonable responsiveness.

	UNROLLING		INLINING	
inlining/unrolling depth n	*Boogie*	*time*	*Boogie*	*time*
3	864	1.07	1201	1.44
4	937	1.13	1822	2.26
5	1010	1.21	3054	4.23
6	1083	1.32	5518	10.93
7	1156	1.52	10446	30.64
8	1229	2.03	20302	112.32
10	1375	4.26	–	–
13	1594	37.52	–	–
15	1667	253.30	–	–

Table 4. Scalability of unrolling and inlining on examples 3 and 6 from Table 3.

7 Related Work

The steadily growing interest for techniques and tools that make verification more approachable indicates how some of the most glaring hurdles to the progress of formal methods lie in their applicability. Tools such as Dafny [27], Spec# [2],

VCC [10], ESC/Java2 [11,20], and Why3 [5] define the state of the art in static program verification. Their approaches rely on accurate specifications, which are not easy to write and get right.

One way to ameliorate this situation is inferring specifications automatically using static [8,24,18] or dynamic [14,37,36] techniques. Specifications dynamically inferred are based on a finite number of executions, and hence may be unsound; this makes them unsuitable for use in the context of static verification. Static techniques can infer sound specifications from the program text; these are useful to document existing implementations, to discover auxiliary assertions (such as loop invariants), or for comparison with specifications independently written, but proving an implementation correct against a specification inferred from it is mostly a vacuous exercise.

The simple implicit contracts that we use in our approach express well-formedness properties of the input program, which are tacitly assumed by programmers reasoning informally about it; therefore, there is no risk of circularity. Some static verifiers use mathematical integers or assume purity of specification functions to have well-formedness by construction; a risk is that, when they are applied to real programming languages, the corresponding semantic gap may leave some errors go unnoticed. ESC/Java2, for example, does not check for overflows [21], nor if specification expressions are executable (for example, null-dereferencing could happen when evaluating a precondition). The Dafny verifier [27] checks well-formedness of pre- and postconditions, and may consequently require users to add explicit contracts to satisfy well-formedness. Our implicit contracts are instead added and checked automatically, without requiring users to explicitly write them. In this sense, they are similar to approaches such as VCC [10], which models the semantics of the C programming language as precisely as possible.

Besides inferring specifications, another approach to facilitate formal verification is combining complementary verification techniques. CEGAR model-checking [3], for example, uses model-checking exhaustive verification techniques on approximate program models, combined with a form of symbolic execution to determine whether the failed verification attempts are indicative of real implementation errors or only a figment of an imprecise abstraction. Tools such as DSD-Crasher [12] and our EVE [33] integrate testing and static checking to find when the errors reported by the latter are spurious. Collaborative verification [9] is also based on the combination of testing and static verification, and on the explicit formalization of the restrictions of each tool used in the combination. Two-step verification also integrates the results of different techniques, with the main purpose of improving error reporting and reducing the number of annotations needed, rather than complementing the limitations of specific techniques.

The Spec# system includes a verification debugger [25] to inspect error models when verification fails; more recently, an interpreter for Boogie programs [29] can help find the sources of failed verification attempts. Debuggers for verification can be quite useful in practice, but achieve a lesser degree of automation

than two-step verification, since users need to manually inspect and understand the error models using the debugger.

Inlining and unrolling are standard techniques in compiler construction. The Boogie verifier [26] also supports inlining of procedures: through annotations, one can require to inline a procedure to a given depth using different sound or unsound definitions. Boogie also supports (unsound) loop unrolling on request. AutoProof's current implementation of inlining and unrolling works at source code level, rather than using Boogie's similar features, to have greater flexibility in how inlining and unrolling are defined and used. Methods specified using the Java Modeling Language (JML) with the "helper" modifier [11] are meant to be used privately; ESC/Java inlines calls to such methods [17]. ESC/Java also unrolls loops a fixed amount of times; users can choose between performing sound or unsound variants of the unrolling. In previous work [28], we used unrolling and inlining to check the type correctness of JavaScript programs. In two-step verification, we use inlining and unrolling completely automatically: users need not be aware of them to benefit from an improved feedback that narrows down the sources of failed verification attempts.

Acknowledgements. Work partially supported by the ERC grant CME/291389; by the SNF grants LSAT (200020-134974) and ASII (200021-134976); and by the Hasler foundation on related projects.

References

1. B. Weide et al. Incremental benchmarks for software verification tools and techniques. In *VSTTE*, pages 84–98. Springer, 2008.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
3. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.
4. J. Bloch. `http://goo.gl/sWLty`, 2006.
5. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie*, pages 53–64, 2011.
6. T. Bormer et al. The COST IC0701 verification competition. In *FoVeOos*, LNCS, pages 3–21. Springer, 2012.
7. P. Chalin. Are practitioners writing contracts? In *RODIN Book*, volume 4157 of *LNCS*, pages 100–113. Springer, 2006.
8. B.-Y. E. Chang and K. R. M. Leino. Inferring object invariants. *ENTCS*, 131:63–74, 2005.
9. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.
10. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
11. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.

12. C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM TOSEM*, 17(2):8, 2008.
13. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
14. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.
15. H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. The evolution of contracts. `http://arxiv.org/abs/1211.4775`, 2012.
16. J.-C. Filliâtre, A. Paskevich, and A. Stump. The 2nd verified software competition: Experience report. In *COMPARE*, volume 873, pages 36–49, 2012.
17. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
18. C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, volume 6300 of *LNCS*, pages 277–300. Springer, 2010.
19. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis verification competition. `http://verifythis2012.cost-ic0701.org`, 2012.
20. P. R. James and P. Chalin. Faster and more complete extended static checking for the Java Modeling Language. *J. Autom. Reasoning*, 44(1-2):145–174, 2010.
21. J. R. Kiniry, A. E. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In *SAVCBS*, pages 19–24. ACM, 2006.
22. V. Klebanov et al. The 1st verified software competition: Experience report. In *FM*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
23. D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 2011.
24. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.
25. C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger. In *SEFM*, LNCS, pages 407–414. Springer, 2011.
26. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.
27. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR-16*, pages 348–370. Springer-Verlag, 2010.
28. M. Nordio, C. Calcagno, and C. A. Furia. Javanni: A verifier for JavaScript. In *FASE*, volume 7793 of *LNCS*, pages 231–234. Springer, 2013.
29. N. Polikarpova. Boogaloo. `http://goo.gl/YH9QT`, 2012.
30. N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104, 2009.
31. N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *ICSE*, pages 257–266. ACM, 2013.
32. C. Tinelli. Formal methods in software engineering. `http://www.divms.uiowa.edu/~tinelli/181/`, 2011.
33. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, volume 7041 of *LNCS*, pages 382–398. Springer, 2011.
34. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Verifying Eiffel programs with Boogie. In *BOOGIE workshop*, 2011. `http://arxiv.org/abs/1106.4700`.
35. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Automatic verification of advanced object-oriented features: The AutoProof approach. In *Tools for Practical Software Verification*, volume 7682 of *LNCS*, pages 134–156. Springer, 2012.
36. A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Autom. Softw. Eng.*, 18(3-4):263–292, 2011.
37. Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, pages 191–200. ACM, 2011.