Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

AutoProof Meets Some Verification Challenges

Julian Tschannen · Carlo A. Furia · Martin Nordio

Abstract AutoProof is an automatic verifier for functional properties of programs written in Eiffel. This paper illustrates some of AutoProof's capabilities when tackling the three challenges of the VerifyThis verification competition held at FM 2012, as well as on three other problems proposed in related events. AutoProof's design focuses on making it practically applicable with reduced user effort. Tackling the challenges demonstrates to what extent this design goal is met in the current implementation: while some of AutoProof's current limitations prevent us from verifying the complete specification of the prefix sum and binary search tree algorithms, we can still prove some partial properties on interesting special cases, but with the advantage of requiring little or no specification.

1 Verification Benchmarks Can Shape the Field

For better or worse, benchmarks shape a field [16]. Patterson's compelling analysis of the coming of age of computer architecture seems to fit the progress of formal software verification too – possibly with a couple-of-decade time shift. As verification techniques left the realm of pure theory and became implementable and usable, they often reported incomparable results: different tools that work on different languages and solve different problems (such as extended static checking, functional correctness, shape analysis, and so on).

Verification competitions and challenges [9, 4, 5, 7] can help in this regard: by providing benchmarks for verification techniques and tools, they help assess progress, compare different approaches, and reward incremental, yet practically relevant, advancements. Hopefully, this will also lead to an outcome similar to computer architecture's: *when a field has good benchmarks, we settle debates and the field makes rapid progress* [16].

This paper assesses the capabilities of AutoProof [22, 21], a static verifier for programs written in Eiffel, describing its capabilities at the time of writing based on solving the challenges of the VerifyThis competition [7] held at FM 2012.[1]

AutoProof – whose fundamental features are presented in Section 2 – still is largely work-in-progress, and hence has some significant limitations compared to more mature verification environments (such as VeriFast [8], Dafny [11], or Why3 [3], to mention just a few). However, rather than replicating the functionalities of other similar verifiers, AutoProof's design focuses on offering improved usability to programmers with relatively little background in formal techniques. Its performance with the three challenges of VerifyThis partly demonstrates this underlying goal: while AutoProof can completely solve only the first challenge, it provides partial interesting solutions to the second and third challenges, but with the advantage of requiring little or no specification compared to that needed for full-fledged proofs.

From the perspective of its developers, tackling verification competition challenges has also been quite useful to highlight some important limitations and subtle shortcomings of the current implementation. The next development steps – some of which are already underway – will benefit from this improved understanding.

J. Tschannen · C. A. Furia · M. Nordio
ETH Zurich
Chair of Software Engineering
Clausiusstrasse 59
8092 Zurich
E-mail: firstname.lastname@inf.ethz.ch

[1] AutoProof was not used during the actual VerifyThis competition, but only later on the competition's problems.

Sections 3–5 describe how AutoProof fares on the three VerifyThis challenges. Section 6 describes other related challenges which highlight more peculiarities of AutoProof.

2 AutoProof

AutoProof is an automatic verifier of functional properties working on Eiffel programs. It is available as an online command-line tool usable without installation, as well as integrated in EVE [20], the Eiffel Verification Environment IDE, distributed as free software and available for download. Visit the homepage of AutoProof for detailed information (and its latest features):

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

The source code of the verification challenges presented here is also available online, and can be verified with AutoProof's online version:

`http://se.inf.ethz.ch/research/autoproof/sttt`

AutoProof is written in Eiffel.

AutoProof uses Boogie [10] as backend: it translates Eiffel programs to Boogie programs, invokes the Boogie verifier on the latter, and traces the outcome back to the corresponding Eiffel code. AutoProof is completely automatic and can verify individual routines, classes, or entire applications. Figure 1 shows a screenshot of AutoProof in EVE, showing a detailed report of a verification attempt on the problem discussed in Section 3. Items in the report can be expanded to display more information, such as which assertions failed, or which attempts were successful only under simplifying assumptions.

2.1 What AutoProof Can Do

AutoProof has extended support of Eiffel language features. Besides the fundamental procedural language constructs (such as routines and routine calls), it fully supports reasoning about objects with polymorphic assignments and dynamic binding. It also supports a methodology to prove programs using agents [14] (which are Eiffel's function objects).

In terms of basic types, AutoProof treats integers as machine integers, with modular arithmetic and checking for possible overflows consistently with their runtime semantics. It has limited support of floating-point numbers, which are translated to rationals in Boogie (infinite precision).

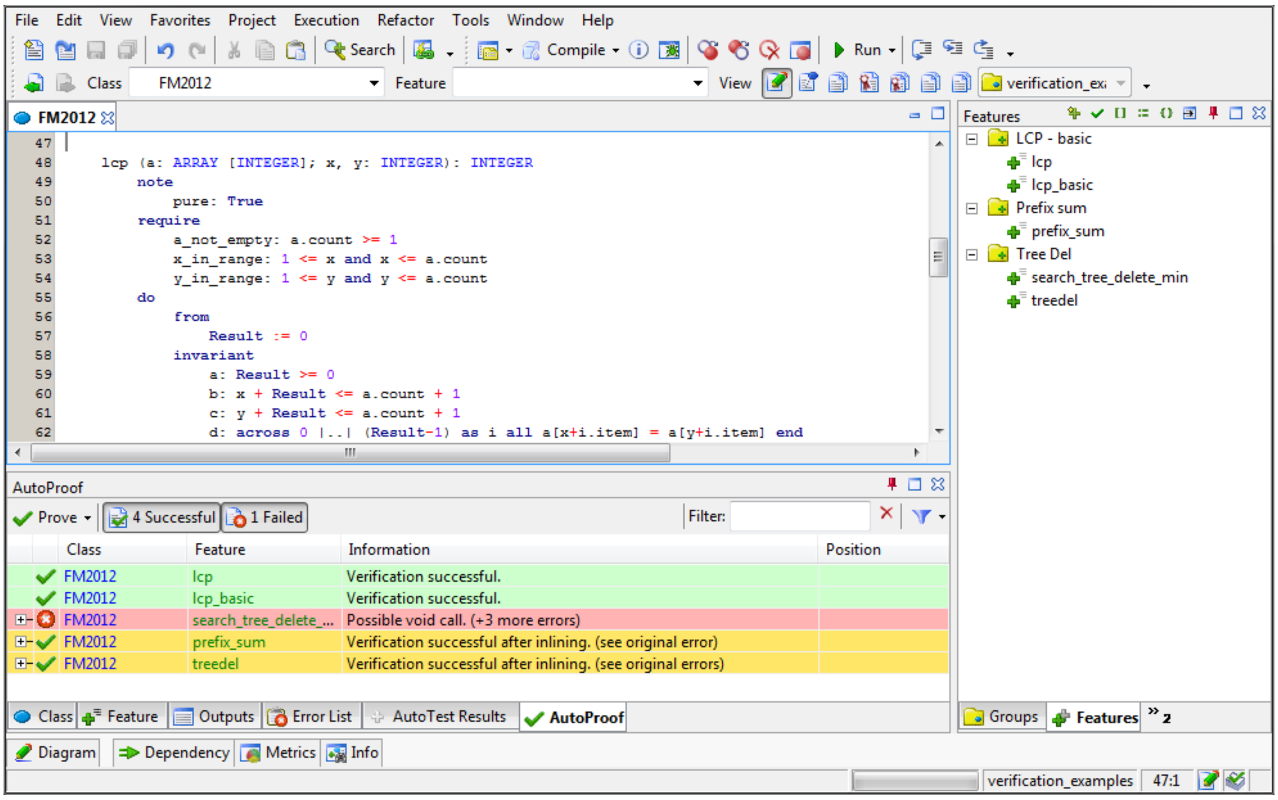
The Eiffel language natively supports contracts (pre- and postconditions and class invariants), which consists of executable assertions used to specify program behavior and are checked at runtime. AutoProof translates Eiffel contracts into Boogie specification elements. Since regular routines can be used in Eiffel contracts, whereas Boogie procedures cannot directly appear in annotations, AutoProof also performs some checks of well-formedness of routines used in contracts. In particular, it checks that they conform to the notion of *purity* [13]: for every routine r appearing in contracts, AutoProof generate verification conditions that encode that executing r produces no effects on the heap; such verification conditions become part of r's proof obligations, which are discharged as part of r's normal verification process.

We recently improved the usability of AutoProof with *two-step verification* [23], a technique that provides detailed user feedback on failed verification attempts. The basic idea of two-step verification is to perform two attempts for each item to be verified; the first attempt follows standard modular reasoning techniques, and the second attempt inlines calls to routines and unrolls loops to some finite bound.[2] Since we can reason about inlined calls and unrolled loops with less specification (using their actual implementations instead), the second verification attempt may succeed in cases where the first modular one fails. These cases suggest that the implementation is likely correct, but the specification is not accurate enough to successfully prove it. Users can then decide to improve the specification to match the implementation, or to simply use the inlined code to prove a routine correct on some specific inputs. As we demonstrate in the rest of the paper, this makes for successful partial verifications that require very few annotations; and generally provides better feedback, suggesting whether the problems with failed verification attempts are in the specification or the implementation. Two-step verification is also useful in cases where we run up against limitations in the underlying reasoning capabilities of the prover, where verification may fail even if specifications are perfectly adequate in principle. We presented the details of two-step verification elsewhere [23]; the present paper focuses on how two-step verification can be used in practice on verification challenges.

2.2 What AutoProof Cannot Do

Only a few features of the Eiffel language are currently not supported by AutoProof, most notably expanded types (with semantics based on values rather than refer-

[2] AutoProof performs inlining while translating Eiffel to Boogie. Within AutoProof's architecture, this solution offers more flexibility than directly using Boogie's inlining feature.



```

    lcp (a: ARRAY [INTEGER]; x, y: INTEGER): INTEGER
        note
            pure: True
        require
            a_not_empty: a.count >= 1
            x_in_range: 1 <= x and x <= a.count
            y_in_range: 1 <= y and y <= a.count
        do
            from
                Result := 0
            invariant
                a: Result >= 0
                b: x + Result <= a.count + 1
                c: y + Result <= a.count + 1
                d: across 0 |..| (Result-1) as i all a[x+i.item] = a[y+i.item] end
```

	Class	Feature	Information	Position
✓	FM2012	lcp	Verification successful.	
✓	FM2012	lcp_basic	Verification successful.	
✗	FM2012	search_tree_delete_...	Possible void call. (+3 more errors)	
✓	FM2012	prefix_sum	Verification successful after inlining. (see original error)	
✓	FM2012	treedel	Verification successful after inlining. (see original errors)	

Fig. 1 Screenshot of the EVE IDE integrating AutoProof. The bottom pane shows a verification report for the code in the top pane (discussed in Section 3). The first two features (in green) are successfully verified; the third feature (in red) failed verification; the fourth and fifth features (in yellow) are verified but only under restricting assumptions (as in the two-step verification technique described in Section 2). One can click on the yellow and red items to get more detailed error reports and suggestions on how to fix the problems.

ences, similar to value types in C#); and strings. We developed a translation [22] for exceptions (whose semantics [15] is different compared to other object-oriented languages), but we have not implemented it yet in AutoProof.

A major limitation follows from the fact that Eiffel does not support annotations for *framing* and lacks a complete methodology to reason about class invariants in the presence of dependencies between different objects [2]. Consequently, AutoProof currently only supports proofs that do not require complex frame specifications – in which case they can be generated automatically based on the items mentioned in postconditions. Adequate support for framing and class invariants is the next major milestone for AutoProof; some preliminary work has laid the groundwork for it [18,17].[3]

Finally, AutoProof is limited in terms of features to interact with the backend prover and to formulate background theories or prove intermediate lemmas necessary for verification. It currently uses a library of Boogie background theories, including axiomatizations of items such as arrays, the heap, and machine integers, but there is no mechanism to express and manage domain-specific theories at the level of the input programming language. Providing such mechanisms is another major priority in the ongoing development of AutoProof.

2.3 Using Boogie as Backend

While AutoProof's architecture is extensible to support multiple backend provers, the current implementation depends on Boogie as an intermediate verification layer. Boogie is a language for verification [10], as well as an automated verifier that takes programs written in the Boogie language as input.

AutoProof verifies Eiffel programs annotated with contracts by encoding their semantics into the Boogie language. Eiffel routines map to Boogie procedures that explicitly operate on the heap encoded as a mapping of references to allocated primitive values. Eiffel contracts become annotations in Boogie's typed first-order logic. Consistency between the Eiffel and the Boogie representations is guaranteed by a background theory: a col-

[3] At the time of finalizing this article, we have completed a flexible methodology for class invariants [19] in AutoProof.

lection of axioms, predicate definitions, and global variables which we wrote as part of AutoProof and which defines the correct semantics of Eiffel within Boogie. For example, Eiffel *INTEGER* variables become **int** variables in Boogie, but the background theory constraints such variables so that they possess the features of machine integers (e.g., they are bounded) rather than those of mathematical integers (which Boogie's **int** type represents).

After translating a routine and its specification to Boogie, AutoProof verifies it by calling the Boogie verifier on the translation. At this point, Boogie is entirely responsible of generating verification conditions and interfacing with a solver (Z3 by default) to discharge them. Boogie's output feeds back to AutoProof, which presents it in the context of the original Eiffel program as shown in Figure 1. This is possible because AutoProof's translation from Eiffel to Boogie also includes structured machine-generated comments, useful for debugging of the translation as well as to trace the verification outcome in Boogie – and in particular failed verification attempts – back to the source Eiffel program that has been translated. Details about AutoProof's Boogie translation are presented in related work [22].

3 Longest Common Prefix

The first VerifyThis challenge is a *longest common prefix* algorithm: given an integer array a and two indexes x and y within its bounds, determine the length of the longest common prefix starting at positions x and y (that is the length of the maximal subarrays from x and y). Consider, for example, the integer array $\ll 1, 2, 3, 4, 1, 2, 3 \gg$ and the indexes[4] 1 and 5 within it: the longest common prefix is the sequence $\ll 1, 2, 3 \gg$ of length 3. For the same array, the longest common prefix for indexes 1 and 3 is the empty sequence because the elements at positions 1 and 3 differ; and the longest common prefix for indexes 1 and 1 is obviously the whole array (of length 7).

Figure 2 shows an Eiffel implementation of the longest common prefix algorithm, as a routine *lcp* fully annotated with precondition (**require**), postcondition (**ensure**), loop **invariant** and loop **variant** (also called "ranking function"). The contracts consist of implicitly conjoined assertions; each assertion may have a label, such as *a_in_range* on line 5. AutoProof can automatically verify this implementation against its specification. Specifically, it proves that, for inputs satisfying the precondition, the postcondition holds when the routine terminates, the loop invariant is inductive, the loop terminates, all array accesses are valid, and there are no integer overflows. We now look into these aspects in detail.

[4] We assume arrays numbered from one, as is the norm in Eiffel.

3.1 Functional Correctness

The postcondition specifies the functional correctness of *lcp* by describing three characterizing properties that the returned integer **Result** must satisfy to represent correct output:

- *in_range*: the output **Result** defines valid subarrays at x and y.
- *is_prefix* : the two subarrays $a[x{:}x+\mathbf{Result}-1]$ and $a[y{:}y+\mathbf{Result}-1]$ of length **Result** starting at x and y are pairwise identical. This postcondition uses Eiffel's **across**..**all** syntax equivalent to the universal quantification $\forall i \in [0..\mathbf{Result}-1] : a[x+i] = a[y+i]$ over the finite integer range $[0..\mathbf{Result}-1]$.
- *longest_prefix* : the two subarrays of length **Result** starting at x and y are maximal; that is, either one of them runs until a's end or the next pair of characters after the subarrays differ. This postcondition uses Eiffel's **or else** short-circuited disjunction.

The specification is completed by the loop invariant: its first three components (*inv1*, *inv2*, *inv3*) are necessary to establish the postcondition *in_range*, and its last component *inv4* to establish *is_prefix* . Postcondition *longest_prefix* follows from the exit conditions on lines 17–19. Notice the peculiar structure of Eiffel loops: the **from** clauses is evaluated once, as if it were regular code appearing before the loop (it is just syntactic sugar); the exit condition in the **until** clause is evaluated before every iteration; correspondingly, the loop body (**loop** clause) may be executed zero times or more.

Our initially unsuccessful attempts at verifying *lcp* prompted us to introduce an improvement in the Boogie translation which is more amenable to automated reasoning with Boogie. The original translation rendered *inv4* roughly as follows:

$$\forall\ i\colon \mathbf{int}\ \bullet\ (0 \le i \land i \le \mathit{Result}-1) \implies (\mathit{Heap}[a,\ x+i]\ =\ \mathit{Heap}[a,\ y+i])$$

where *Heap* is a mapping representing fields allocated in the heap. Boogie cannot establish that this assertion implies the translation of *is_prefix* , even if the two assertions are identical in Eiffel (and hence in Boogie). The problem traced back to using the arithmetic operation + to adding a logic variable and a global variable (the problem does not occur when we add a variable

```
 1   lcp (a: ARRAY [INTEGER]; x, y: INTEGER): INTEGER
 2       note
 3         pure
 4       require
 5           a_in_range : 1 ≤ a.count and a.count < {INTEGER}.max_value
 6           x_in_range : 1 ≤ x and x ≤ a.count
 7           y_in_range : 1 ≤ y and y ≤ a.count
 8       do
 9           from
10               Result := 0
11           invariant
12               inv1: Result ≥ 0
13               inv2: x + Result ≤ a.count + 1
14               inv3: y + Result ≤ a.count + 1
15               inv4: across 0 |..| (Result−1) as i all a[x+i] = a[y+i] end
16           until
17               x + Result = a.count + 1 or else
18               y + Result = a.count + 1 or else
19               a[x+Result] ≠ a[y+Result]
20           loop
21               Result := Result + 1
22           variant
23               a.count − Result
24           end
25       ensure
26           in_range : (Result ≥ 0) and
27                      (x + Result ≤ a.count + 1) and
28                      (y + Result ≤ a.count + 1)
29           is_prefix : across 0 |..| (Result−1) as i all a[x+i] = a[y+i] end
30           longest_prefix : (x + Result = a.count + 1) or else
31                            (y + Result = a.count + 1) or else
32                            (a[x+Result] ≠ a[y+Result])
33       end
```

Fig. 2 Implementation of the *lcp* algorithm.

to a numeric constant). We solved the problem by introducing Boogie logic functions wrapping arithmetic operations within the scope of quantifiers, such as

$$\textbf{function}\ \mathit{add}(a,\ b\colon \textbf{int})\colon\ \textbf{int}\ \{\ a + b\ \}$$

for addition. The Boogie translation of the loop invariant *inv4* simply becomes

$$\forall\ i\colon\ \textbf{int}\ \bullet\ (0 \le i \wedge i \le \mathit{Result}{-}1) \Longrightarrow (\mathit{Heap}[a,\ \mathit{add}(x,\ i)] = \mathit{Heap}[a,\ \mathit{add}(y,\ i)])$$

which Boogie can reason about without difficulties.

This trick does not affect the semantics of the translation or what properties can be expressed, but was necessary to accommodate a peculiarity of Boogie's behavior, namely that instantiation triggers cannot include interpreted symbols like the plus sign [12]. This is a recurring scenario for tool developers whose implementations depend on others' tools.

3.2 Framing

As mentioned in Section 2, AutoProof currently supports only limited forms of framing specifications, and has to resort to special annotations because Eiffel does not have constructs to natively express framing. The framing specification for routine *lcp* is, however, quite simple: the routine does not modify any global variable nor allocates new objects and is therefore (strongly) *pure* [13]. Line 3 specifies this using a **note** clause (similar to C#'s attributes or Java's annotations). AutoProof processes **note** clauses and verifies that the routine is indeed pure, that is side-effect free.

3.3 Array Accesses

Using the precondition and the loop invariants which restrict the range of the two index variables to always be in the range of the array, AutoProof also verifies that all array accesses are within a's bounds. This entails, in particular, that predicates involving arrays used in the specification are well-formed; the loop's exit condition, for example, evaluates the last disjunct only if the first two evaluate to false (**or else** is short-circuited), which implies that x + **Result** and y + **Result** are in bounds.

3.4 Integer Overflows

AutoProof has an option, enabled by default, to verify that no arithmetic operations may overflow. Preconditions *x_in_range* and *y_in_range* specify that *x* and *y* are in bounds, but this is not enough to guarantee that no overflow occurs: the index of the last element of an array with size the largest machine integer *max_value* is the value 1 + *max_value* (array indexing starts at 1 in Eiffel), which produces an overflow. Thus, precondition *a_in_range* restricts the size of the array to less than the maximum integer value. Under this additional precondition, AutoProof verifies that there are no integer overflows.

3.5 Termination

AutoProof uses the loop variant on line 23 to prove termination of the loop. It also checks that the variant is a valid variant, that is it decreases after every loop iteration, and has a lower bound (determined in this case by the size of the array *a.count*).

3.6 Clients

In addition to verifying the *lcp* routine against its specification, we can use AutoProof to check client code that calls *lcp*. For example, the following test cases initialize an array with seven integer values (using the Eiffel syntax ≪ ... ≫), call *lcp* on the array with different values for *x* and *y*, and assert (**check** in Eiffel) that the results are correct.

```
local
  a: ARRAY [INTEGER]
do
  a := ≪ 1, 2, 3, 4, 1, 2, 3 ≫
  check lcp (a, 1, 5) = 3 end
  check lcp (a, 2, 6) = 2 end
  check lcp (a, 1, 1) = 7 end
  check lcp (a, 1, 3) = 0 end
end
```

Even if all assertions are logical consequences of *lcp*'s postcondition, the Boogie translation produced by AutoProof fails to verify the last one. At this point, two-step verification (see Section 2) makes an additional attempt consisting of inlining *lcp*'s body at the call site while ignoring its postcondition; since this attempt successfully verifies all calls, AutoProof feedback suggests that *lcp*'s implementation is correct (after all, it even satisfies its postcondition), but the way the postcondition is written prevents the prover from succeeding.

We find a quick fix consisting of adding an assertion that explicitly mentions a special fact about the array values:

```
check a[1 + 0] ≠ a[3 + 0] end
```

The translation of this new assertion acts as a trigger to instantiate quantifiers, which Boogie passes on to Z3 and makes verification of the following assertion succeed. The assertion $a[1 + 0] \neq a[3 + 0]$ may be placed at any point in the client before the assertion where the trigger is necessary, since Boogie collects all assertions it has encountered so far. Based on this additional explicit piece of information, Boogie realizes that there are no valid instantiations of the quantifier, and hence the result must be 0.

Since it may be hard for the client to anticipate the need for such an additional assertion, we suggest generalizing it into a postcondition of *lcp*:[5]

```
(Result = 0) = (a[x] ≠ a[y])
```

which does not affect the specification of the routine but makes it more readily usable to verify arbitrary clients. With this postcondition, Boogie also verifies calls to *lcp* that return 0 without the need to suggest quantifier instantiations.

4 Prefix Sum

The second VerifyThis challenge is a *prefix sum* algorithm: given an array *a* of integers, construct an array *b* of the same length, such that *b*'s *k*th element *b*[*k*] equals the sum $\sum_{1 \leq i < k} a[i]$ of all elements in *a* at positions up to *k* excluded. For example, if *a* is $\ll 1, 2, 3, 4 \gg$, *b* is $\ll 0, 1, 3, 6 \gg$.

The challenging aspect of the VerifyThis algorithm is that it is based on a version of prefix sum that computes *b in place*, that is by directly modifying *a* without allocating a fresh array. This is achieved in two passes called *upsweep* and *downsweep*. For simplicity, assume that *a*'s size is a power of two, so that we can identify the array elements with the leaves of a complete binary tree. The upsweep phase propagates the sum of the children up to the parents in the tree, with as many rounds as the tree's height; for example, $\ll 1, 2, 3, 4 \gg$ first becomes $\ll 1, \mathbf{3}, 3, \mathbf{7} \gg$ and then $\ll 1, 3, 3, \mathbf{10} \gg$ (we mark in bold the elements modified in each round). The downsweep starts by putting a zero in the rightmost array cell (corresponding to the tree's root); then, each node passes its value to its left child, and passes the sum of the left child's overwritten value and its own

[5] The operator = represents both equality and double implication (for Booleans).

```
array: ARRAY [INTEGER]

upsweep: INTEGER
 local
   space, left, right: INTEGER
 do
   from
     space := 1
   until
     space ≥ array.count
   loop
     from
       left := space
     until
       left > array.count
     loop
       right := left + space
       array[right] := array[left] + array[right]
       left := left + space ∗ 2
     end
     space := space ∗ 2
   end
   Result := space
 end

downsweep (a_space: INTEGER)
 local
   space, left, right, temp: INTEGER
 do
   space := a_space
   array[array.count] := 0
   from
     space := space // 2    −− integer division
   until
     space ≤ 0
   loop
     from
       right := space ∗ 2
     until
       right > array.count
     loop
       left := right − space
       temp := array[right]
       array[right] := array[left] + array[right]
       array[left] := temp
       right := right + space ∗ 2
     end
     space := space // 2    −− integer division
   end
 end
```

Fig. 3 Implementation of the iterative prefix-sum algorithm without any specification.

value to its right child. Continuing the example, start from $\ll 1, 3, 3, \mathbf{0} \gg$, which becomes $\ll 1, \mathbf{0}, 3, \mathbf{3} \gg$ and then $\ll \mathbf{0}, \mathbf{1}, \mathbf{3}, \mathbf{6} \gg$, which is the correct solution. Figure 3 shows an implementation of this algorithm, as two routines *upsweep* and *downsweep* working iteratively on a global array variable *array*.

4.1 Modular Verification

Some restrictions of the Eiffel language for assertions prevents us from readily expressing the specification of *upsweep* and *downsweep*. Specifically, **old** expressions (which refer to the value of their argument at the routine's entry) cannot be used in loop invariants, but any reasonable specification of *upsweep* and *downsweep*'s loops must express how the original *array* is modified at each iteration. Using an implementation based on recursion, rather than iteration, would not solve the problem, as **old** expressions are also forbidden in postconditions within the scope of bounded **across** quantifiers.[6] Even though it might be possible to use workarounds, such as making copies of arrays in the beginning of the routine which can then be used instead of **old** expressions or by defining specific predicates for each property, these workarounds make the specification and subsequent verification too difficult at the moment or hinder the flexibility of the developer.

[6] The rationale for these rules is to simplify the runtime checking of assertions; they are, however, unnecessarily restricting for specifications meant for static verification.

4.2 Client Verification

Even if *upsweep* and *downsweep* have no specification, AutoProof can verify concrete clients by *inlining* the routines' bodies in the client code. Consider, for example, the following test case with an array of size eight:

```
local
  p: PREFIX_SUM_ITER
  space: INTEGER
do
  a := ≪ 3, 1, 7, 0, 4, 1, 6, 3 ≫
  create p.make (a)
  space := p.upsweep
  check p.array = ≪ 3, 4, 7, 11, 4, 5, 6, 25 ≫
  p.downsweep (space)
  check p.array = ≪ 0, 3, 4, 11, 11, 15, 16, 22 ≫
end
```

AutoProof deploys two-step verification (see Section 2) and reports a "conditional" success (such as the lines highlighted in yellow in Figure 1): the assertions in the test case are verified only using inlining and exhaustively unrolling the loops. This suggests that the implementation is probably correct (at least with respect to the usage done in the given client), but the specification is insufficient to generalize the proof to a modular setting. Our experience suggests that this kind of feedback is generally quite useful: it helps speedup the debugging of failed verification attempts and it guides the generalization of partial verification attempts.

4.3 Arrays of Fixed Size

We can take advantage of a simplifying suggestion mentioned in VerifyThis's challenge, consisting of assuming that the algorithm only work on arrays of eight elements. Under this assumption, we can prove *upsweep* and *downsweep* correct using AutoProof by generalizing what is done in the previous subsection. First, we add the simplifying assumption $array.count = 8$ explicitly as a precondition of both *upsweep* and *downsweep*. Then, we exhaustively specify their postconditions for arrays of size eight; *upsweep*'s postcondition is then:

```
ensure
    array[1] = old(array[1])
    array[2] = old(array[1] + array[2])
    array[3] = old(array[3])
    array[4] = old(array[1] + array[2] + array[3] +
                   array[4])
    array[5] = old(array[5])
    array[6] = old(array[5] + array[6])
    array[7] = old(array[7])
    array[8] = old(array[1] + array[2] + array[3] +
                   array[4] + array[5] + array[6] +
                   array[7] + array[8])
end
```

Routine *downsweep*'s postcondition is similar, but expressing the final result; and *downsweep*'s precondition also has to include *upsweep*'s postcondition, since the former is correct only when called after the latter. With this setup, AutoProof verifies the implementation in Figure 3 by exhaustively unrolling the loops. This corresponds to a complete verification for arrays of size eight without requiring to write any loop invariant.

4.4 Other Properties

As in the problem of Section 3, AutoProof also verifies that all array accesses are in bounds, and establishes termination (trivially, since the loops iterate $3 = \log_2 8$ times). AutoProof can also verify generic clients that conform to the simplifying precondition (arrays of size eight) using modular reasoning.

Finally, AutoProof verifies absence of overflows under the additional precondition that:

```
across 1 |..| 8 as i all
    −100000000 ≤ array[i] ≤ 100000000
end
```

```
search_tree_delete_min (old_root : TREE_NODE):
        [new_root: TREE_NODE; min: INTEGER]
 local
    tt, pp, p, new_root: TREE_NODE
    min: INTEGER
 do
    p := old_root . left
    if p = Void then
      min := old_root . data
      new_root := old_root . right
    else
      from
        pp := old_root
        tt := p. left
      until
        tt = Void
      loop
        pp := p
        p := tt
        tt := p. left
      end
      min := p.data
      tt := p. right
      pp. set_left (tt)
      new_root := old_root
    end
    Result := [new_root, min]
 end
```

Fig. 4 Implementation of the tree-deletion algorithm.

5 Binary Search Tree: Deletion

The third VerifyThis challenge is an algorithm to delete the node containing the minimum value in binary search trees. To this end, it is sufficient to traverse the tree starting from the root, always visiting the left child of the current node until a leaf is found; the leaf stores the minimum value by construction.

Figure 4 shows an implementation of the algorithm in Eiffel, following the outline given in the VerifyThis competition. Routine *search_tree_delete_min* returns the deleted minimum value, as well as a new root node (which changes if the minimum is stored in the root); the two returned values are packed in a *tuple* (denoted by square brackets in Eiffel). The implementation operates on objects of class *TREE_NODE*, whose definition is outlined in Figure 5 for reference (some inessential details are omitted for simplicity).

The current limitations on the expressible assertions prevent us from formalizing the complete routine specification in a form that AutoProof can process. Specifically, the gist of the specification requires expressing

```
class TREE_NODE

  make (l, r: TREE_NODE; v: INTEGER)
    require
      left_smaller : l ≠ Void implies l.data ≤ v
      right_larger : r ≠ Void implies r.data ≥ v
    do
      left := l
      right := r
      data := v
    end

  left : TREE_NODE
  right : TREE_NODE
  data: INTEGER

end
```

Fig. 5 Implementation of the *TREE_NODE* class.

the sequence of values stored in the nodes and visited during the loop, as well as to formalize the impact of modifying a leaf node on the whole tree structure (that is, framing). We are working on supporting such specifications in AutoProof based on the notion of model-based contracts [17,18], but, at the time of writing, AutoProof cannot verify the tree algorithm in the general case.

However, as we did for the previous problems, we can still verify given client code by using the exhaustive inlining and unrolling capabilities of AutoProof, activated automatically by two-step verification. For example, consider the following code fragment creating a tree with three nodes and deleting the minimum.

```
treedel_client (a, b, c: INTEGER)
  require
    a < b and b < c
  local
    node1, node2, node3: TREE_NODE
    res: [root: TREE_NODE; min: INTEGER]
  do
    create node3.make (Void, Void, a)
    create node2.make (Void, Void, c)
    create node1.make (node3, node2, b)
    res := search_tree_delete_min (node1)
    check res.root = node1 end
    check res.min = b end
  end
```

The concluding **check** instructions assert that the two components of the returned tuple are indeed the root (unchanged in this case) and the minimum value. While this does not meet the full challenge, it is interesting that AutoProof can verify the algorithm for all trees with a given structure and size without requiring any specification but only based on the semantics of instructions.

6 Other Challenges

We briefly present solutions to challenges offered by other verification competitions, to further demonstrate AutoProof's current capabilities, specification style, and limitations. All three challenges consist of algorithms working on arrays; since, as we repeatedly remarked in the paper, AutoProof's main current limitation is the lack of complete support for framing specifications, algorithms working on linked data structures are largely beyond its current capabilities.

6.1 Maximum in an Array

The COST 2011 competition [4] required to verify an algorithm that finds the maximum element in unsorted arrays. The twist is the requirement to perform a two-way search, which scans the array content from both ends and terminates when the two indexes meet in the middle. Figure 6 shows an annotated implementation in Eiffel which AutoProof can verify; routine *max_in_array* returns the index of the largest element.

The peculiarity of the implementation reflects on the structure of the loop invariant, which includes two symmetric **across** quantifications over the left-hand side and the right-hand side of the array scanned so far. Also notice that postcondition *is_max* uses the **across** quantifier in a different way, since it directly quantifies over array elements rather than over integer ranges as done in previous examples. As usual, AutoProof also verifies that all array accesses are in bounds and that the loop terminates. Verifying the absence of overflows requires a stronger precondition that the array length is less than the largest machine integer (see Section 3.4).

6.2 Sum and Max

The VSTTE 2010 competition [9] required to verify an algorithm that computes the sum and the maximum element of an array. The specification to be proven is not complete, but only asserts that the maximum value times the array length is an upper bound on the sum. Figure 7 shows an Eiffel implementation with the given specification, which AutoProof can verify; routine *sum_and_max* returns a tuple with the values for sum and max.

```
 1 max_in_array (a: ARRAY [INTEGER]): INTEGER
 2  note
 3    pure
 4  require
 5    not_empty: a.count > 0
 6  local
 7    x, y: INTEGER
 8  do
 9    from
10      x := 1
11      y := a.count
12    invariant
13      1 ≤ x and x ≤ y and y ≤ a.count
14      across 1 |..| x as i all
15              a[i] ≤ a[x] or a[i] ≤ a[y] end
16      across y |..| a.count as i all
17              a[i] ≤ a[x] or a[i] ≤ a[y] end
18    until
19      x = y
20    loop
21      if a[x] ≤ a[y] then
22        x := x + 1
23      else
24        y := y − 1
25      end
26    variant
27      y − x
28    end
29    Result := x
30  ensure
31    in_range: 1 ≤ Result and Result ≤ a.count
32    is_max: across a as e all e ≤ a[Result] end
33  end
```

Fig. 6 Implementation of the *max in array* challenge of COST 2011.

```
 1 sum_and_max (a: ARRAY [INTEGER]):
 2       [sum, max: INTEGER]
 3  note
 4    pure_fresh
 5  require
 6    not_empty: a.count > 0
 7    not_negative : across a as j all j ≥ 0 end
 8  local
 9    i, sum, max: INTEGER
10  do
11    from
12      i := 1
13    invariant
14      1 ≤ i and i ≤ a.count + 1
15      across 1 |..| (i−1) as j all a[j] ≤ max end
16      sum ≤ (i−1) ∗ max
17      across a as j all j ≥ 0 end
18    until
19      i > a.count
20    loop
21      sum := sum + a[i]
22      if a[i] > max then
23        max := a[i]
24      end
25      i := i + 1
26    variant
27      a.count − i + 1
28    end
29    Result := [sum, max]
30  ensure
31    sum_in_range: Result.sum ≤ a.count ∗ Result.max
32  end
```

Fig. 7 Implementation of the *sum & max* challenge of VSTTE 2010.

Two aspects of this algorithm are interesting from the perspective of automated verification. First, the assertions use nonlinear arithmetic (that is, multiplication of variables). Boogie has some support for integer multiplication and division, which AutoProof leverages in the translation to render Eiffel's semantics of integer operations.

The second interesting aspect is framing, specified using a **note** annotation (line 4) as done in Section 3.2. The frame specification for *sum_and_max* is *pure_fresh*, which denotes routines that do not modify the object state but may allocate and change fresh objects (this notion is also called weak purity [13]). In fact, line 29 implicitly creates a new tuple [7] to store the result.

6.3 Two-way Sort

The VSTTE 2012 competition [5] required to verify an algorithm that sorts Boolean arrays in linear time. The algorithm scans the input array from both ends, looking for and swapping pairs of inverted elements. It is a technique similar to Dijkstra's Dutch flag algorithm [6] but working on the two Boolean values rather than on the three flag colors. The specification is that the array is sorted when the algorithm terminates.

Our initial solution directly used arrays containing *BOOLEAN* values, which AutoProof translates to arrays of **bool** in Boogie. Unfortunately, Boogie cannot reason efficiently about Boolean arrays, and in fact it could not prove this initial solution even if it was correct. Therefore, we tried another solution: using arrays of integer type constrained to contain only the values 0 and 1. This solution, shown in Figure 8, is much more amenable to Boogie's automatic reasoning, and in fact AutoProof can prove its correctness without difficulties. The precondition on line 4 and the loop invariant on line 15 restrict the input array to only contain values 0 and 1; establishing the other components required for functional correctness depends on this restriction.

An interesting aspect which demonstrates the capabilities of AutoProof is the usage of a separate routine *swap* called to switch inverted elements. Standard verifiers leverage modular reasoning, which entails that

[7] *TUPLE* is a reference type in Eiffel.

```
 1 two_way_sort (a: ARRAY [INTEGER])
 2   require
 3     not_empty: a.count ≥ 0
 4     boolean: across a as k all k = 0 or k = 1 end
 5   local
 6     i, j: INTEGER
 7   do
 8     from
 9       i := 1
10       j := a.count
11     invariant
12       i ≥ 1 and i ≤ j + 1 and j ≤ a.count
13       across a as k all k = 0 or k = 1 end
14       across 1 |..| (i−1) as k all a[k] = 0 end
15       across (j+1) |..| a.count as k all a[k] = 1 end
16     until
17       i ≥ j
18     loop
19       if a[i] = 0 then
20         i := i + 1
21       elseif a[j] = 1 then
22         j := j − 1
23       else
24         swap (a, i, j)
25         i := i + 1
26         j := j − 1
27       end
28     variant
29       j − i + 1
30     end
31   ensure
32     sorted: across 1 |..| (a.count−1) as k all
33                       a[k] ≤ a[k+1] end
34   end
35
36 swap (a: ARRAY [INTEGER]; i, j: INTEGER)
37   note
38     inline_in_caller
39   local
40     t: INTEGER
41   do
42     t := a[i]
43     a[i] := a[j]
44     a[j] := t
45   end
```

Fig. 8 Implementation of the *two-way sort* challenge of VSTTE 2012 using integer arrays.

the effect of a routine call within the caller is limited to what is mentioned in the callee's specification (its postcondition, in particular). Therefore, verification of implementations such as that in Figure 8 would fail because *swap* has no specification, and hence its effect within *two_way_sort* is undetermined. AutoProof, however, supports two-way verification: after a first unsuccessful attempt at modular verification, it tries to inline *swap*'s body within *two_way_sort* and notices that verification is successful in this case. AutoProof reports such "partially successful" attempts in yellow in the GUI (see Figure 1), and offers two options: either just use *swap* inlined whenever it is called, or provide a suitable specification to *swap* so that the correctness proof can be carried out modularly. In this simple example, where *swap* is just a helper function and writing its complete specification seems an overkill, we opted for the first option: we added the annotation *inline_in_caller* on line 38, which makes AutoProof inline *swap* whenever necessary without complaining about its lack of specification. This reduces the specification burden on users, thus making the whole verification a bit more practical.

7 Conclusions & Lessons Learned

The VerifyThis 2012 challenges were a valuable testbed for AutoProof. They showed that the tool has finally reached a stage where it can completely verify nontrivial problems such as the first challenge. They also allowed us to demonstrate some features of two-step verification, which can provide informative feedback and reduce the amount of annotation required in simple cases. In particular, Sections 4.3 and 5 discussed how we verified the second and third challenges in special cases with little or no specification. Even if their complete verification is still beyond AutoProof's current capabilities, it is within the reach of the planned improvements to the tool.

Assessing AutoProof's limitations was also valuable and useful to define its future development agenda in detail. It is clear that the major feature that is lacking is a comprehensive methodology to specify abstractly and to reason about complex object structures and about framing. This will be the next major improvement introduced in AutoProof.

Lessons Learned. Generalizing from our experience with AutoProof, there are some lessons learned which emphasize the complex trade-offs involved in turning formal verification into something practical.

The first lesson follows from the observation – obvious in hindsight – that an automatic verifier such as AutoProof, which aims at working on a real full-fledged object-oriented programming language, is implemented as a component deeply embedded in a long and complex toolchain, which encompasses parsers, translators, and multiple intermediate language representations and abstraction levels (the compiler's, Boogie's, the underlying SMT solver's, etc.). Such a deeply layered structure brings great advantages in terms of reuse: AutoProof does not have to know, for instance, the details of Eiffel's multiple inheritance resolution, since it gets the results of the compiler's analysis; or how to best generate and encode verification conditions in a form amenable to the underlying SMT solver, since Boogie takes care of that. At the same time, a long toolchain is often inflexible, as it introduces fundamental dependencies which may be burdensome to remove, as well as limitations that are difficult to overcome. For example, a few details of AutoProof's encoding into Boogie have been hacked out based on trials and errors and accommodate some specific idiosyncrasies of Microsoft Research's verification system. Removing AutoProof's dependency on Boogie could be useful to avail of other verification engines with possibly complementary characteristics, but is not going to happen anytime soon since it would likely require a major reengineering effort largely divergent from AutoProof's primary development goals.

Another general point is whether one should aim at completely supporting a real programming language or, if the goal is advancing the state of the art of program verification, focus on simpler (possibly ad hoc) languages that are easier to implement and reason about. This is also a complex trade-off, which involves multiple local optima. On the one hand, supporting a full programming language requires dealing with many annoyingly low-level details, which may seem to bring us away from the ultimate goal of improving verification techniques and methods. In fact, this was one of the major lessons reported by the Spec# team [1]. On the other hand, if we want to make verification practical and within the capabilities of programmers who are not fluent in formal techniques, our tools should be interoperable with the same tools the programmers already use in their run-of-the-mill programming, such as IDEs, compilers, and debuggers. Therefore, we should be able to handle most programming language constructs that are used in practice, or at least have well-defined boundaries between what can and cannot be specified and verified.

We are looking forward to these developments and to new verification challenges that will help us assess the achieved progress.

Acknowledgements Work partially supported by SNF grants LSAT/200020-134974, ASII/200021-134976; by Hasler-Stiftung grant #2327; and by ERC grant CME/291389.

References

1. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the spec# experience. *Commun. ACM*, 54(6):81–91, June 2011.
2. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *In MPC, volume 3125 of LNCS*, pages 54–84. Springer, 2004.
3. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie*, pages 53–64, 2011.
4. T. Bormer et al. The COST IC0701 verification competition. In *FoVeOos*, LNCS, pages 3–21. Springer, 2012.
5. J.-C. Filliâtre, A. Paskevich, and A. Stump. The 2nd verified software competition: Experience report. In *COMPARE*, pages 36–49, 2012.
6. D. Gries. *The science of programming*. Springer-Verlag, 1981.
7. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis verification competition. `http://verifythis2012.cost-ic0701.org`, 2012.
8. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of APLAS 2010*, pages 304–311. Springer, 2010.
9. V. Klebanov et al. The 1st verified software competition: Experience report. In *FM*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
10. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.
11. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR-16*, pages 348–370. Springer, 2010.
12. K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In S. Y. Shin and S. Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*, pages 615–622. ACM Press, 2009.
13. D. A. Naumann. Observational purity and encapsulation. In *FASE*, volume 3442 of *LNCS*, pages 190–204. Springer, 2005.
14. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In *Proceedings of TOOLS-EUROPE*, volume 6141 of *LNCS*, pages 79–96. Springer, 2010.
15. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE*, volume 33, pages 195–214, 2009.
16. D. Patterson. For better or worse, benchmarks shape a field: technical perspective. *Commun. ACM*, 55(7):104–104, July 2012.
17. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *Proceedings of VSTTE'10*, volume 6217 of *LNCS*, pages 127–141. Springer, 2010.
18. N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *ICSE*, pages 257–266. ACM, 2013.
19. N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer. Flexible invariants through semantic collaboration. `http://arxiv.org/abs/1311.6329`, November 2013.
20. 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.
21. 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`.
22. 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.
23. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Program checking with less hassle. In *VSTTE*, volume 8164 of *LNCS*, pages 149–169. Springer, 2014.