AutoProof: Auto-active Functional Verification of Object-oriented Programs

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova

Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland
`firstname.lastname@inf.ethz.ch`

Abstract. Auto-active verifiers provide a level of automation intermediate between fully automatic and interactive: users supply code with annotations as input while benefiting from a high level of automation in the back-end. This paper presents AutoProof, a state-of-the-art auto-active verifier for object-oriented sequential programs with complex functional specifications. AutoProof fully supports advanced object-oriented features and a powerful methodology for framing and class invariants, which make it applicable in practice to idiomatic object-oriented patterns. The paper focuses on describing AutoProof's interface, design, and implementation features, and demonstrates AutoProof's performance on a rich collection of benchmark problems. The results attest AutoProof's competitiveness among tools in its league on cutting-edge functional verification of object-oriented programs.

1 Auto-active Functional Verification of Object-oriented Programs

Program verification techniques differ wildly in their degree of automation and, correspondingly, in the kinds of properties they target. One class of approaches—which includes techniques such as abstract interpretation and model checking—is fully *automatic* or "push button", the only required input being a program to be verified; to achieve complete automation, these approaches tend to be limited to verifying simple or implicit properties such as absence of invalid pointer dereference. At the other end of the spectrum are *interactive* approaches to verification—which include tools such as KeY [3]—where the user is ultimately responsible for providing input to the prover on demand, whenever it needs guidance through a successful correctness proof; in principle, this makes it possible to verify arbitrarily complex properties, but it is approachable only by highly-trained verification experts.

In more recent years a new class of approaches have emerged that try to achieve an intermediate degree of automation in the continuum that goes from automatic to interactive—hence their designation [22] as the portmanteau *auto-active*[1]. Auto-active tools need no user input during verification, which proceeds autonomously until it succeeds or fails; however, the user is still expected to provide guidance indirectly through *annotations* (such as loop invariants) in the input program. The auto-active approach has the potential to better support *incrementality*: proving simple properties would require little annotations and of the simple kinds that novice users may be able to provide;

[1] Although *inter-matic* would be as good a name.

proving complex properties would still be possible by sustaining a heavy annotation burden.

This paper describes AutoProof, an auto-active verifier for functional properties of (sequential) object-oriented programs. In its latest development state, AutoProof offers a unique combination of features that make it a powerful tool in its category and a significant contribution to the state of the art. AutoProof targets a real complex object-oriented programming language (Eiffel)—as opposed to more abstract languages designed specifically for verification. It supports most language constructs, as well as a full-fledged verification methodology for heap-manipulating programs based on a flexible annotation protocol, sufficient to completely verify a variety of programs that are representative of object-oriented idioms as used in practice. AutoProof was developed with extensibility in mind: its annotation library can be augmented with new abstract models, and its implementation can accommodate changes in the input language. While Eiffel has a much smaller user base than other object-oriented languages such as C++, Java, and C#, the principles behind AutoProof are largely language independent; hence, they are relevant to a potentially large number of researchers and users—for whom this paper is written.

The verification challenges we use to evaluate AutoProof (Sect. 5) are emerging as the gold standard [17] to demonstrate the capabilities of program provers for functional correctness which, unlike fully automatic tools, use different formats and conventions for input annotations and support specifications of disparate expressiveness, and hence cannot directly be compared on standard benchmark implementations.

Previous work of ours, summarized in Sect. 2.2, described the individual techniques available in AutoProof. This paper focuses on presenting AutoProof's functionalities (Sect. 3), on describing significant aspects of its design and implementation (Sect. 4), and on outlining the results of experiments with realistic case studies, with the goal of showing that AutoProof's features and performance demonstrate its competitiveness among other tools in its league—auto-active verifiers for object-oriented programs.

AutoProof is available as part of the open-source Eiffel Verification Environment (EVE) as well as online in your browser; the page

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

contains source and binary distributions, detailed usage instructions, a user manual, an interactive tutorial, and the benchmarks solutions discussed in Sect. 5.

2 Related Work

2.1 Program Verifiers

In reviewing related work, we focus on the tools that are closer to AutoProof in terms of features, and design principles and goals. Only few of them are, like AutoProof, auto-active, work on real object-oriented programming languages, and support the verification of general functional properties. Krakatoa [10] belongs to this category, as it works on Java programs annotated with a variant of JML (the Java Modeling Language [18]). Since it lacks a full-fledged methodology for class invariants and framing, using Krakatoa to verify object-oriented idiomatic patterns—such as those we discuss in Sect. 5.1—would be quite impractical; in fact, the reference examples distributed with Krakatoa target the verification of algorithmic problems where object-oriented features are immaterial. Similar observations apply to the few other auto-active tools working on Java and JML, such as ESC/Java2 [5] or the more recent OpenJML [25,7]. Even when ESC/Java2 was used on industrial-strength case studies (such as the KOA e-voting system [16]), the emphasis was on modeling and correct-by-construction development, and verification was normally applied only to limited parts of the systems. By contrast, the Spec# system [1] was the forerunner in a new research direction, also followed by AutoProof, that focuses on the complex problems raised by object-oriented structures with sharing, object hierarchies, and collaborative patterns. Spec# works on an annotation-based dialect of the C# language and supports an ownership model which is suitable for hierarchical object structures, as well as visibility-based invariants to specify more complex object relations. Collaborative object structures as implemented in practice (Sect. 5.1) require, however, more flexible methodologies [27] not currently available in Spec#. Tools, such as VeriFast [15], based on separation logic provide powerful methodologies through abstractions different than class invariants, which may lead to a lower level of automation than tools such as AutoProof and a generally higher annotation overhead—ultimately targeting highly trained users.

The experience with the Spec# project suggested that targeting a real object-oriented programming language introduces numerous complications and may divert the focus away from fundamental problems in tool-supported verification. The Dafny program verifier [21] was developed based on this lesson: it supports a simple language expressly designed for verification, which eschews most complications of real object-oriented programming languages (such as inheritance and a complex memory model). Other auto-active verifiers target programming language paradigms other than object orientation. Leon [30] and Why3 [11], for example, work on functional programming languages—respectively, a subset of Scala and a dialect of ML; VCC [6] works on C programs and supports object invariants but with an emphasis on memory safety of low-level concurrent code.

AutoProof lies between automatic and interactive tools in the wide spectrum of verification tools. The CodeContract checker (formerly known as Clousot [24]) is a powerful static analyzer for .NET languages that belongs to the former category (and hence it is limited to properties expressible in its abstract domains). The KeY system [3] for Java belongs to the latter category: while it supports SMT solvers as back-ends to automatically discharge some verification conditions, its full-fledged usage requires explicit user interactions to guide the prover through the verification process.

2.2 Our Previous Work on AutoProof

In previous work, we formalized some critical object-oriented features as they are available in Eiffel, notably function objects (called "agents" in Eiffel) and inheritance and polymorphism [33]. An important aspects for usability is reducing annotation overhead; to this end, we introduced heuristics known as "two-step verification" [34] and demonstrated them on algorithmic challenges [31]. We recently presented the theory behind AutoProof's invariant methodology [27], which includes full support for class invariants, framing, and ghost code. The current paper discusses how these features are available in AutoProof, with a focus on advanced object-oriented verification challenges.

3 Using AutoProof

AutoProof is a static verifier for Eiffel programs which interacts with users according to the auto-active paradigm [22]: verification attempts are completely automated ("push button"), but users are expected in general to provide additional information in the form of *annotations* (loop invariants, intermediate assertions, etc.) for verification to succeed.

AutoProof targets the verification of functional correctness. Given a collection of Eiffel classes, it tries to establish that: routines satisfy their pre/post and frame specifications and maintain class invariants; routine calls take place in states satisfying the callee's precondition; loops and recursive calls terminate; integer variables do not overflow; there are no dereferences of **`Void`** (**`null`**) objects.

AutoProof's techniques are *sound*[2]: successful verification entails that the input program is correct with respect to its given specification. Since it deals with expressive specifications, AutoProof is necessarily *incomplete*: failed verification may indicate functional errors but also shortcomings of the heuristics of the underlying theorem prover (which uses such heuristics to reason in practice about highly-complex and undecidable logic fragments).

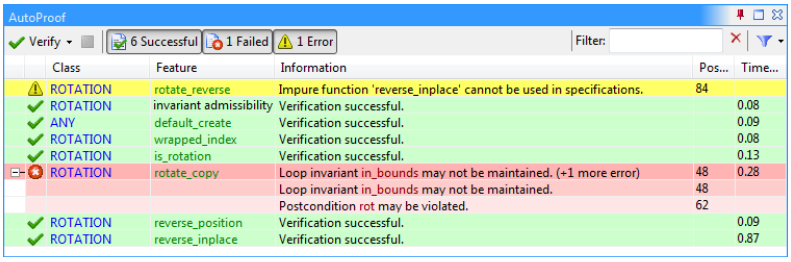
Dealing with inconclusive error reports in incomplete tools is a practical hurdle to usability that can spoil user experience—especially for novices. To improve user feedback in case of failed verification attempts, AutoProof implements a collection of heuristics known as "*two-step verification*" [34]. When they are enabled, each failed verification attempt is transparently followed by a second step that is in general unsound (as it uses under-approximations such as loop unrolling) but helps discern whether failed verification is due to real errors or just to insufficiently detailed annotations. Users see the combined output from the two steps in the form of suggestions to improve the program and its annotations. For example, if verification of a loop fails in the first step but succeeds with finite loop unrolling, the suggestion is that there are no obvious errors in the loop but the loop invariant should be strengthened to make it inductive.

3.1 User Interface (UI)

AutoProof offers its core functionalities both through a command line interface (CLI) and a library (API). End users normally interact with AutoProof through one of two graphical interfaces (GUI): a web-based GUI is available at `http://cloudstudio.ethz.ch/comcom/#AutoProof`; and AutoProof is fully integrated in EVE, the open-source research branch of the EiffelStudio development environment. The following presentation focuses on AutoProof in EVE, but most features are available in every UI.

Users launch AutoProof on the current project, or on specific classes or members thereof. Verification proceeds in the background until it terminates, is stopped, or times

[2] As usual, modulo bugs in the implementation.



	Class	Feature	Information	Pos...	Time...
⚠	ROTATION	rotate_reverse	Impure function 'reverse_inplace' cannot be used in specifications.	84	
✓	ROTATION	invariant admissibility	Verification successful.		0.08
✓	ANY	default_create	Verification successful.		0.09
✓	ROTATION	wrapped_index	Verification successful.		0.08
✓	ROTATION	is_rotation	Verification successful.		0.13
⊟ ⊗	ROTATION	rotate_copy	Loop invariant in_bounds may not be maintained. (+1 more error)	48	0.28
			Loop invariant in_bounds may not be maintained.	48	
			Postcondition rot may be violated.	62	
✓	ROTATION	reverse_position	Verification successful.		0.09
✓	ROTATION	reverse_inplace	Verification successful.		0.87

Fig. 1: The AutoProof output panel showing verification results in EVE.

out. Results are displayed in a panel such as in Fig. 1: each entry corresponds to a routine of some class and is colored to summarize verification outcome. Green entries are successfully verified; red entries have failed verification; and yellow entries denote invalid input, which cannot be translated and verified (for example, impure functions with side effects used as specification elements determine invalid input). Red entries can be expanded into more detailed error messages or suggestions to fix them (when enabled, two-step verification helps provide more precise suggestions). For example, the failed verification entry for a routine may detail that its loop invariant may not be maintained, or that it may not terminate; and suggest that the loop invariant be strengthened, or a suitable variant be provided.

AutoProof's UI is deliberately kept simple with few options and sensible defaults. For advanced users, fine-grained control over AutoProof's behavior is still possible through program annotations, which we outline in the next section.

3.2 Input Language Support

AutoProof supports most of the Eiffel language as used in practice, obviously including Eiffel's native notation for contracts (specification elements) such as pre- and postconditions, class invariants, loop invariants and variants, and inlined assertions such as **`check`** (**`assert`** in other languages). Object-oriented features—classes and types, multiple inheritance, polymorphism—are fully supported [33], and so are imperative and procedural constructs.

Partially supported and unsupported features. A few language features that AutoProof does not currently fully support have a semantics that violates well-formedness conditions required for verification: AutoProof doesn't support specification expressions with side effects (for example, a precondition that creates an object). It also doesn't support the semantics of **`once`** routines (similar to **`static`** in Java and C#), which would require global reasoning thus breaking modularity.

Other partially supported features originate in the distinction between machine and mathematical representation of types. Among primitive types, machine `INTEGER`s are fully supported (including overflows); floating-point `REAL`s are modeled as infinite-precision mathematical reals; strings are not supported but for single-character operations. Array and list library containers with simplified interfaces are supported out of the box. Other container types require custom specification; we recently developed a fully verified full-fledged data structure library including sets, hash tables, and trees [9].

Agents (function objects) are partially supported, with some restrictions in their specifications [33]. The semantics of native **`external`** routines is reduced to their specification. We designed [33] a translation for exceptions based on the latest draft of the Eiffel language standard, but AutoProof doesn't support it yet since the Eiffel compiler still only implements the obsolete syntax for exceptions (and exceptions have very limited usage in Eiffel anyway).

Annotations for verification. Supporting effective auto-active verification requires much more than translating the input language and specification into verification conditions. AutoProof supports *semantic collaboration*, a full-fledged framing methodology we designed to reason about class invariants of structures made of collaborating objects, integrated with a standard *ownership* model; both are described in detail in our previous work [27]. AutoProof's verification methodology relies on annotations that are not part of the Eiffel language. Annotations in assertions or other specification elements use predefined dummy features with empty implementation. Annotations of this kind include *modify* and *read* clauses (specifying objects whose state may be modified or read by a routine's body). For instance, a clause **`modify`** `(set)` in a routine's precondition denotes that executing the routine may modify objects in `set`.

Annotations that apply to whole classes or features are expressed by means of Eiffel's **`note`** clauses, which attach additional information that is ignored by the Eiffel compiler but is processed by AutoProof. Annotations of this kind include defining class members as *ghost* (only used in specifications), procedures as *lemmas* (outlining a proof using assertions and ghost-state manipulation), and which members of a class define its abstract *model* (to be referred to in interface specifications). For example **`note`** `status: ghost` tags as ghost the member it is attached to.

A distinctive trait of semantic collaboration, as available to AutoProof users, is the combination of flexible expressive annotations with useful defaults. Flexible annotations offer fine-grained control over the visibility of specification elements (for example, invariant clauses can be referenced individually); defaults reduce the amount of required manual annotations in many practical cases. The combination of the two is instrumental in making AutoProof usable on complex examples of realistic object-oriented programs.

Verifier's options. AutoProof *verification options* are also expressed by means of **`note`** clauses: users can disable generating boilerplate implicit contracts, skip verification of a specific class, disable termination checking (only verify partial correctness), and define a custom mapping of a class's type to a Boogie theory file. See AutoProof's manual for a complete list of features, options, and examples of usage.

Specification library. To support writing complex specifications, AutoProof provides a library—called MML for Mathematical Model Library—of pre-defined abstract types . These includes mathematical structures such as sets, relations, sequences, bags (multisets), and maps. The MML annotation style follows the model-based paradigm [26], which helps write abstract and concise, yet expressive, specifications. MML's features are fully integrated in AutoProof by means of effective mappings to Boogie background theories. A distinctive advantage of providing mathematical types as an annotated library is that MML is *extensible*: users can easily provide additional abstractions by writing annotated Eiffel classes and by linking them to background theories using custom **note** annotations—in the very same way existing MML classes are defined. This is not possible in most other auto-active verifiers, where mathematical types for specification are built into the language syntax.

```
binary_search (a: ARRAY [INTEGER]; value: INTEGER): INTEGER
  require sorted: is_sorted (a.sequence)
  local low, up, middle: INTEGER
  do
    from low := 1; up := a.count + 1
    invariant
      low_and_up_range: 1 ≤ low and low ≤ up and up ≤ a.count + 1
      result_range: Result = 0 or 1 ≤ Result and Result ≤ a.count
      not_left: across 1 |..| (low−1) as i all a.sequence[i] < value end
      not_right: across up |..| a.count as i all value < a.sequence[i] end
      found: Result > 0 implies a.sequence[Result] = value
    until low ≥ up or Result > 0
    loop
      middle := low + ((up − low) // 2)
      if a[middle] < value then low := middle + 1
      elseif a[middle] > value then up := middle
      else Result := middle end
    variant (a.count − Result) + (up − low) end
  ensure
    present: a.sequence.has (value) = (Result > 0)
    not_present: not a.sequence.has (value) = (Result = 0)
    found_if_present: Result > 0 implies a.sequence[Result] = value
  end
```

Fig. 2: Binary search implementation verified by AutoProof.

Input language syntax. Fig. 2 shows an example of annotated input: an implementation of binary search (problem BINS in Tab. 1) that AutoProof can verify. From top to bottom, the routine `binary_search` includes signature, precondition (**require**), **local** variable declarations, body consisting of an initialization (**from**) followed by a **loop** that executes **until** its exit condition becomes true, and postcondition (**ensure**). The loop's annotations include loop **invariant** and **variant**. Each specification element consists of clauses, one per line, with a *tag* (such as *sorted* for the lone precondition clause) for identification in error reports. Quantified expressions in contracts use the **across** syntax, which corresponds to (bounded) first-order universal (**across** ... **all**) and existential (**across** ... **some**) quantification. For example, loop invariant clause *not_left* corresponds to $\forall i\colon 1 \leq i < \texttt{low} \Longrightarrow \texttt{a.sequence}[i] < \texttt{value}$.

4 How AutoProof Works: Architecture and Implementation

As it is customary in deductive verification, AutoProof translates input programs into verification conditions (VCs): logic formulas whose validity entails correctness of the input programs. Following the approach pioneered by Spec# [1] and since adopted by numerous other tools, AutoProof does not generate VCs directly but translates Eiffel programs into Boogie programs [20] and calls the Boogie tool to generate VCs from the latter. Boogie is a simple procedural language tailored for verification, as well as a verification tool that takes programs written in the Boogie language, generates VCs for them, feeds the VCs to an SMT solver (Z3 by default), and interprets the solver's output in terms of elements of the input Boogie program. Using Boogie decouples VC generation from processing the source language (Eiffel, in AutoProof's case) and takes advantage of Boogie's efficient VC generation capabilities.

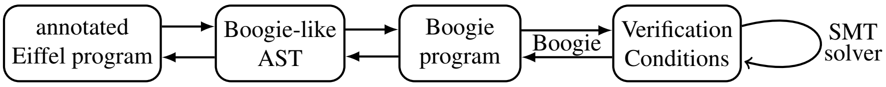


Fig. 3: Workflow of AutoProof with Boogie back-end.

As outlined in Fig. 3, AutoProof implements the translation from Eiffel to Boogie in two stages. In the first stage, it processes an input Eiffel program and translates it into a Boogie-like abstract syntax tree (AST); in the second stage, AutoProof transcribes the AST into a textual Boogie program.

The rest of this section focuses on describing how AutoProof's architecture (Sect. 4.1) and implementation features (Sect. 4.2) make for a flexible and customizable translation process. An extended version of this paper [35] also outlines the mapping from Eiffel to Boogie. We focus on discussing the challenges tackled when developing AutoProof and the advantages of our implemented solutions.

4.1 Extensible Architecture

Top-level API. Class `AUTOPROOF` is the main entry point of AutoProof's API. It offers features to submit Eiffel code, and to start and stop the verification process. Objects of class `RESULT` store the outcome of a verification session, which can be queried by calling routines of the class. One can also register an Eiffel **`agent`** (function object) with an `AUTOPROOF` object; the outcome `RESULT` object is passed to the agent for processing as soon as it is available. This pattern is customary in reactive applications such as AutoProof's GUI in EVE.

Translation to Boogie. An abstract syntax tree (AST) expresses the same semantics as Eiffel source code but using elements reflecting Boogie's constructs. Type relations such as inheritance are explicitly represented (based on type checking) using axiomatic constraints, so that ASTs contain all the information necessary for verification. The transcription of the AST into a concrete Boogie program is implemented by a *visitor* of the AST. Modifying AutoProof in response to changes in Boogie's syntax would only require to modify the visitor.

Extension points. AutoProof's architecture incorporates *extension points* where it is possible to programmatically modify and extend AutoProof's behavior to implement different verification processes. Each extension point maintains a number of *handlers* that take care of aspects of the translation from Eiffel to the Boogie-like AST. Multiple handlers are composed according to the *chain of responsibility* pattern; this means that a handler may only implement the translation of one specific source language element, while delegating to the default AutoProof handlers in all other cases. A new translation feature can thus be added by writing a handler and registering it at an extension point. Extension points target three program elements of different generality.

Across extension points handle the translation of Eiffel **`across`** expressions, which correspond to quantified expressions. Handlers can define a semantics of quantification over arbitrary data structures and domains. (AutoProof uses this extension point to translate quantifications over arrays and lists.)

Call extension points handle the translation of Eiffel calls, both in executable code and specifications. Handlers can define translations specific to certain data types. (AutoProof uses this extension point to translate functions on integers and dummy features for specification.)

Expression extension points handle the translation of expressions. Handlers can define translations of practically every Eiffel expression into a Boogie-like AST representation. This extension point subsumes the other two, which offer a simpler interface sufficient when only specific language elements require a different translation.

The flexibility provided for by extension points is particular to AutoProof: the architecture of other similar tools (Spec#, Dafny, and OpenJML) does not seem to offer comparable architectural features for straightforward extensibility in the object-oriented style.

4.2 Implementation Features

AutoProof's implementation consists of about 25'000 lines of Eiffel code in 160 classes.

Modular translation. AutoProof performs *modular* reasoning: the effects of a call to `p` within routine `r`'s body are limited to what is declared in `p`'s specification (its pre- and postcondition and frame) irrespective of `p`'s body (which is only used to verify `p`'s correctness). To achieve modularity incrementally, AutoProof maintains a *translation pool* of references to Eiffel elements (essentially, routines and their specifications). Initially, it populates the pool with references to the routines of the classes specified as input to be verified. Then, it proceeds as follows: (1) select an element `el` from the pool that hasn't been translated yet; (2) translate `el` into Boogie-like AST and mark `el` as translated; (3) if `el` refers to (i.e., calls) any element `p` not in the pool, add a reference to `p`'s specification to the pool; (4) if all elements in the pool are marked as translated stop, otherwise repeat (1). This process populates the pool with the transitive closure of the "calls" relation, whose second elements in relationship pairs are specifications, starting from the input elements to be verified.

Traceability of results. The auto-active paradigm is based on interacting with users at the high level of the source language; in case of failed verification, reports must refer to the input Eiffel program rather than to the lower level (Boogie code). To this end, AutoProof follows the standard approach of adding structured comments to various parts of the Boogie code—most importantly to every assertion that undergoes verification: postconditions; preconditions of called routine at call sites; loop invariants; and other intermediate **`assert`**s. Comments may include information about the *type* of condition that is checked (postcondition, loop termination, etc.), the *tag* identifying the clause (in Eiffel, users can name each assertion clause for identification), a *line* number in the Eiffel program, the *called* routine's name (at call sites), and whether an assertion was *generated* by applying a default schema that users have the option to disable (such as in the case of default class invariant annotations [27]). For each assertion that fails verification, AutoProof reads the information in the corresponding comment and makes it available in a `RESULT` object to the **`agent`**s registered through the API to receive verification outcomes about some or all input elements. `RESULT` objects also include information about verification times. This *publish/subscribe* scheme provides fine-grained control on how results are displayed.

Bulk vs. forked feedback. AutoProof provides feedback to users in one of two modes. In *bulk* mode all input is translated into a single Boogie file; results are fed back to users when verification of the whole input has completed. Using AutoProof in bulk mode minimizes translation and Boogie invocation overhead but provides feedback synchronously, only when the whole batch has been processed. In contrast, AutoProof's *forked* mode offers asynchronous feedback: each input routine (and implicit proof obligations such as for class invariant admissibility checking) is translated into its own self-contained Boogie file; parallel instances of Boogie run on each file and results are fed back to users asynchronously as soon as any Boogie process terminates. AutoProof's UIs use the simpler bulk mode by default, but offer an option to switch to the forked mode when responsiveness and a fast turnaround are deemed important.

5 Benchmarks and Evaluation

We give capsule descriptions of benchmark problems that we verified using the latest version of AutoProof; the complete solutions are available at `http://se.inf.ethz.ch/research/autoproof/repo` through AutoProof's web interface.

5.1 Benchmarks Description

Our selection of problems is largely based on the verification challenges put forward during several scientific forums, namely the SAVCBS workshops [28], and various verification competitions [17,4,12,14] and benchmarks [36]. These challenges have recently emerged as the customary yardstick against which to measure progress and open challenges in verification of full functional correctness.

Tab. 1 presents a short description of verified problems. For complete descriptions see the references (and [27] for our solutions to problems 11–17). The table is partitioned in three groups: the first group (1–10) includes mainly *algorithmic* problems; the second group (11–17) includes object-oriented design challenges that require complex *invariant* and *framing* methodologies; the third group (18–27) targets *data-structure* related problems that combine algorithmic and invariant-based reasoning. The second and third group include cutting-edge challenges of reasoning about functional properties of objects in the heap; for example, PIP describes a data structure whose node invariants depend on objects not accessible in the physical heap.

5.2 Verified Solutions with AutoProof

Tab. 2 displays data about the verified solutions to the problems of Sect. 5.1; for each problem: the number of Eiffel classes (#C) and routines (#R), the latter split into *gh*ost functions and lemma procedures and *co*ncrete (non-ghost) routines; the lines of executable Eiffel CODE and of Eiffel SPECIFICATION (a total of T specification lines, split into preconditions P, postconditions Q, frame specifications F, loop invariants L and variants V, auxiliary annotations including ghost code A, and class invariants C); the

#	NAME	DESCRIPTION	FROM
1	Arithmetic (ARITH)	Build arithmetic operations based on the increment operation.	[36]
2	Binary search (BINS)	Binary search on a sorted array (iterative and recursive version).	[36]
3	Sum & max (S&M)	Sum and maximum of an integer array.	[17]
4	Search a list (SEARCH)	Find the index of the first zero element in a linked list of integers.	[17]
5	Two-way max (2-MAX)	Find the maximum element in an array by searching at both ends.	[4]
6	Two-way sort (2-SORT)	Sort a Boolean array in linear time using swaps at both ends.	[12]
7	Dutch flag (DUTCH)	Partition an array in three different regions (specific and general verions).	[8]
8	LCP (LCP)	Longest common prefix starting at given positions x and y in an array.	[14]
9	Rotation (ROT)	Circularly shift a list by k positions (multiple algorithms).	[13]
10	Sorting (SORT)	Sorting of integer arrays (multiple algorithms).	
11	Iterator (ITER)	Multiple iterators over a collection are invalidated when the content changes.	[28, '06]
12	Subject/observer (S/O)	Design pattern: multiple observers cache the content of a subject object.	[28, '07]
13	Composite (CMP)	Design pattern: a tree with consistency between parent and children nodes.	[28, '08]
14	Master clock (MC)	A number of slave clocks are loosely synchronized to a master.	[2]
15	Marriage (MAR)	Person and spouse objects with co-dependent invariants.	[23]
16	Doubly-linked list (DLL)	Linked list whose nodes have links to left and right neighbors.	[23]
17	PIP (PIP)	Graph structure with cycles where each node links to at most one parent.	[29]
18	Closures (CLOSE)	Various applications of function objects.	[19]
19	Strategy (STRAT)	Design pattern: a program's behavior is selected at runtime.	[19]
20	Command (CMD)	Design pattern: encapsulate complete information to execute a command.	[19]
21	Map ADT (MAP)	Generic map ADT with layered data.	[36]
22	Linked queue (QUEUE)	Queue implemented using a linked list.	[36]
23	Tree maximum (TMAX)	Find the maximum value in nodes of a binary tree.	[4]
24	Ring buffer (BUFF)	A bounded queue implemented using a circular array.	[12]
25	Hash set (HSET)	A hash set with mutable elements.	
26	Board game 1 (GAME1)	A simple board game application: players throw dice and move on a board.	
27	Board game 2 (GAME2)	A more complex board game application: different board-square types.	

Table 1: Descriptions of benchmark problems.

S/C specification to code ratio (measured in tokens)[3]; the lines of BOOGIE input (where *tr* is the problem-specific translation code and *bg* are the included background theory necessary for verification); the overall verification time (in bulk mode). AutoProof ran on a single core of a Windows 7 machine with a 3.5 GHz Intel i7-core CPU and 16 GB of memory, using Boogie v. 2.2.30705.1126 and Z3 v. 4.3.2 as backends.

Given that we target full functional verification, our specification to code ratios are small to moderate, which demonstrates that AutoProof's notation and methodology support concise and effective annotations for verification. Verification times also tend to be moderate, which demonstrates that AutoProof's translation to Boogie is effective.

To get an idea of the kinds of annotations required, we computed the ratio A/T of auxiliary to total annotations. On average, 2.8 out of 10 lines of specification are auxiliary annotations; the distribution is quite symmetric around its mean; auxiliary annotations are less than 58% of the specification lines in all problems. Auxiliary annotations tend to be lower level, since they outline intermediate proof goals which are somewhat specific to the way in which the proof is carried out. Thus, the observed range of A/T ratios seems to confirm how AutoProof supports incrementality: complex proofs are possible but require more, lower level annotations.

5.3 Open Challenges

The collection of benchmark problems discussed in the previous sections shows, by and large, that AutoProof is a state-of-the-art auto-active tool for the functional verifica-

[3] In accordance with common practices in verification competitions, we count *tokens* for the S/C ratio; but we provide other measures in *lines*, which are more naturally understandable.

#	NAME	#C	#R		CODE	SPECIFICATION								S/C	BOOGIE		TIME [s]
			co	*gh*		*T*	*P*	*Q*	*F*	*L*	*V*	*A*	*C*		*tr*	*bg*	
1	ARITH	1	6	0	99	44	11	12	0	12	9	0	0	0.4	927	579	3.1
2	BINS	1	4	1	62	48	11	12	0	6	3	16	0	1.6	965	1355	3.7
3	S&M	1	1	0	23	12	3	2	1	4	0	2	0	1.0	638	1355	3.9
4	SEARCH	2	5	1	57	62	2	12	2	6	2	27	11	2.3	931	1355	4.1
5	2-MAX	1	1	0	23	12	2	4	0	4	2	0	0	2.3	583	1355	3.0
6	2-SORT	1	2	0	35	28	5	7	2	6	2	6	0	1.8	683	1355	3.2
7	DUTCH	1	4	1	72	75	13	22	4	21	0	15	0	2.6	1447	1355	4.1
8	LCP	2	2	0	40	28	4	7	0	6	2	9	0	1.0	1359	1355	4.2
9	ROT	1	3	3	51	74	14	10	3	17	2	28	0	2.6	1138	1355	4.1
10	SORT	1	9	6	177	219	31	38	9	56	5	80	0	2.6	2302	1355	5.8
11	ITER	3	8	0	88	69	15	26	6	0	0	11	11	1.4	1461	1355	8.9
12	S/O	3	6	0	71	56	10	14	4	3	0	15	10	1.4	1156	1355	4.4
13	CMP	2	5	3	54	125	19	18	5	0	2	72	9	4.3	1327	1355	7.5
14	MC	3	7	0	63	61	9	14	5	0	0	26	7	1.8	956	579	3.7
15	MAR	2	5	0	45	50	12	11	3	0	0	19	5	2.3	755	579	3.3
16	DLL	2	8	0	69	76	12	14	4	0	0	39	7	2.0	891	579	4.4
17	PIP	2	5	1	54	111	23	18	6	0	1	56	7	3.9	988	1355	5.8
18	CLOSE	9	18	0	145	106	40	31	8	0	0	22	5	0.8	2418	688	5.7
19	STRAT	4	4	0	43	5	0	4	0	0	0	1	0	0.2	868	579	3.3
20	CMD	6	8	0	77	32	4	14	2	0	0	10	5	0.7	1334	579	3.3
21	MAP	1	8	0	78	67	6	29	2	6	4	15	5	2.3	1259	1355	4.1
22	QUEUE	4	13	1	121	101	11	26	1	0	0	48	15	1.5	2360	1355	7.4
23	TMAX	1	3	0	31	43	3	12	2	0	2	19	5	2.1	460	1355	3.2
24	BUFF	1	9	0	66	54	8	19	4	0	0	12	11	1.1	1256	1355	4.4
25	HSET	5	14	5	146	341	45	39	10	20	2	197	28	3.7	3546	1355	13.7
26	GAME1	4	8	0	165	93	16	13	4	31	3	10	16	1.2	4044	1355	26.6
27	GAME2	8	18	0	307	173	25	27	11	48	3	29	30	1.4	7037	1355	54.2
	total	72	184	22	2262	2165	354	455	98	246	44	784	184	1.9	43089	1355	203.8

Table 2: Verification of benchmark problems with AutoProof.

tion of object-oriented programs. To our knowledge, no other auto-active verifier fully supports the complex reasoning about class invariants that is crucial to verify object-oriented pattern implementation such as S/O and PIP. It is important to remark that we're describing *practical* capabilities of tools: other auto-active verifiers may support logics sufficiently rich to express the semantics of object-oriented benchmarks, but this is a far cry from automated verification that is approachable idiomatically at the level of a real object-oriented language. Also, AutoProof's performance is incomparable against that of interactive tools, which may still offer some automation but always have the option of falling back to asking users when verification gets stuck.

The flip side of AutoProof's focus on supporting a real object-oriented language is that it may not be the most powerful tool to verify purely algorithmic problems. The benchmarks have shown that AutoProof still works quite well in that domain, and there are no intrinsic limitations that prevent from applying it to the most complex examples. However, algorithmic verification is often best approached at a level that abstracts from implementation details (such as pointers and objects) and can freely use high-level constructs such as infinite maps and nondeterminism. Verifiers such as Dafny [21] and Why3 [11], whose input languages have been explicitly designed to match such abstraction level, are thus best suited for algorithmic verification, which is instead not the primary focus of AutoProof.

Another aspect of the complexity vs. expressivity trade-off emerges when verifying realistic data structure implementations (or, more generally, object-oriented code as it is written in real-life projects). Tools such as Dafny offer a bare-bones framing methodology that is simple to learn (and to teach) and potentially very expressive; but it becomes unwieldy to reason about complicated implementations, which require to deal with an abundance of special cases by specifying each of them at a low level of detail—and annotational complexity easily leads to unfeasible verification. AutoProof's methodology is richer, which implies a steeper learning curve but also a variety of constructs and defaults that can significantly reduce the annotational overhead and whose custom Boogie translation offers competitive performance in many practical cases.

Given AutoProof's goal of targeting a real programming language, there are few domain-specific features of the Eiffel language that are not fully supported but are used in practice in a variety of programs: reasoning in AutoProof about strings and floating-point numbers is limited by the imprecision of the verification models of such features. For instance (see Sect. 3.2), floating point numbers are translated as infinite-precision reals; precise reasoning requires manually specifying properties of floating point operations. Another domain deliberately excluded from AutoProof so far is concurrent programming. As a long term plan, we envision extending AutoProof to cover these domains to the extent possible: precise functional verification of such features is still largely an open challenge for automated verification tools.

A related goal of AutoProof's research is verifying a fully-specified realistic data structure library—the first such verification carried out entirely with an auto-active tool. This effort—one of the original driving forces behind designing AutoProof's features—has been recently completed with the verification of the EiffelBase2 container library [9].

6 Discussion

How do AutoProof's techniques and implementation generalize to other domains? While Eiffel has its own peculiarities, it is clear that AutoProof's techniques are applicable with little changes to other mainstream object-oriented languages such as Java and C#; and that AutoProof's architecture uses patterns that lead to proper designs in other object-oriented languages too.

A practically important issue is the input language, namely how to reconcile the conflicting requirements of supporting Eiffel as completely as possible and of having a convenient notation for expressing annotations necessary for auto-active verification. While Eiffel natively supports fundamental specification elements (pre- and postconditions and invariants), we had to introduce ad hoc notations, using naming conventions and dummy features, to express modifies clauses, ghost code, and other verification-specific directives in a way that is backward compatible with Eiffel syntax. We considered different implementation strategies, such as using a pre-processor or extending Eiffel's parser, but we concluded that being able to reuse standard Eiffel tools without modifying them is a better option in terms of reusability and compatibility (as the language and its tools may evolve), albeit it sacrifices a bit of notational simplicity. This trade-off is reasonable whenever the goal is verifying programs in a real language used in practice; verifiers focused on algorithmic challenges would normally prefer ad hoc notations with an abstraction level germane to the tackled problems.

In future work, AutoProof's architecture could integrate translations to back-end verifiers other than Boogie. To this end, we could leverage verification systems such as

Why3 [11], which generates verification conditions and discharges them using a variety of SMT solvers or other provers.

Supporting back-ends with different characteristics is one of the many aspects that affect the *flexibility* of AutoProof and similar tools. Another crucial aspect is the quality of feedback in case of failed verification attempts, when users have to change the input to fix errors and inconsistencies, work around limitations of the back-end, or both. As mentioned in Sect. 3, AutoProof incorporates heuristics that improve feedback. Another component of the EVE environment combines AutoProof with automatic random testing and integrates the results of applying both [32]. As future work we plan to further experiment with integrating the feedback of diverse code analysis tools (AutoProof being one of them) to improve usability of verification.

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, 2011. `http://specsharp.codeplex.com/`.
2. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
4. T. Bormer et al. The COST IC0701 verification competition 2011. In *FoVeOOS*, volume 7421 of *LNCS*, pages 3–21. Springer, 2012. `http://foveoos2011.cost-ic0701.org/verification-competition`.
5. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, LNCS, pages 342–363. Springer, 2006. `http://kindsoftware.com/products/opensource/ESCJava2/`.
6. 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. `http://vcc.codeplex.com/`.
7. D. Cok. The OpenJML toolset. In *NASA Formal Methods*, volume 6617. 2011.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
9. EiffelBase2: A fully verified container library. `https://github.com/nadia-polikarpova/eiffelbase2`, 2015.
10. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007. `http://krakatoa.lri.fr/`.
11. J.-C. Filliâtre and A. Paskevich. Why3 – where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013. `http://why3.lri.fr/`.
12. J.-C. Filliâtre, A. Paskevich, and A. Stump. The 2nd verified software competition: Experience report. In *COMPARE*, volume 873 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012. `https://sites.google.com/site/vstte2012/compet`.
13. C. A. Furia. Rotation of sequences: Algorithms and proofs. `http://arxiv.org/abs/1406.5453`, June 2014.
14. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis verification competition. `http://verifythis2012.cost-ic0701.org`, 2012.
15. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010. `http://people.cs.kuleuven.be/~bart.jacobs/verifast/`.

16. J. R. Kiniry, A. E. Morkan, D. Cochran, F. Fairmichael, P. Chalin, M. Oostdijk, and E. Hubbers. The KOA remote voting system: A summary of work to date. In *TGC*, volume 4661 of *LNCS*, pages 244–262. Springer, 2007.
17. V. Klebanov et al. The 1st verified software competition: Experience report. In *FM*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011. https://sites.google.com/a/vscomp.org/main/.
18. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.
19. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
20. K. R. M. Leino. This is boogie 2. Technical report, Microsoft Research, June 2008. http://research.microsoft.com/apps/pubs/default.aspx?id=147643.
21. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010. http://research.microsoft.com/en-us/projects/dafny/.
22. K. R. M. Leino and M. Moskal. Usable auto-active verification. In *Usable Verification Workshop*. http://fm.csl.sri.com/UV10/, November 2010.
23. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.
24. F. Logozzo. Our experience with the CodeContracts static checker. In *VSTTE*, volume 7152 of *LNCS*, pages 241–242. Springer, 2012. http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.
25. The OpenJML toolset. http://openjml.org/, 2013.
26. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *VSTTE*, volume 6217 of *LNCS*, pages 127–141. Springer, 2010.
27. N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer. Flexible invariants through semantic collaboration. In *FM*, volume 8442 of *LNCS*, pages 514–530. Springer, 2014.
28. SAVCBS workshop series. http://www.eecs.ucf.edu/~leavens/SAVCBS/, 2010.
29. A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *IWACO*, pages 1–9. ACM, 2009.
30. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011. http://leon.epfl.ch/.
31. J. Tschannen, C. A. Furia, and M. Nordio. AutoProof meets some verification challenges. *International Journal on Software Tools for Technology Transfer*, pages 1–11, February 2014.
32. 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.
33. 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 133–155. Springer, 2012.
34. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Program checking with less hassle. In *VSTTE 2013*, volume 8164 of *LNCS*, pages 149–169. Springer, 2014.
35. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. http://arxiv.org/abs/1501.03063, 2015.
36. B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *VSTTE*, number 5295 in LNCS, pages 84–98. Springer, 2008.