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

AutoProof: auto-active functional verification of object-oriented programs*

Carlo A. Furia[1], Martin Nordio[2], Nadia Polikarpova[3], Julian Tschannen[2]**

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] Chair of Software Engineering, ETH Zurich, Zurich, Switzerland
[3] MIT CSAIL, Cambridge, MA, USA

The date of receipt and acceptance will be inserted by the editor

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.

Key words: Functional verification · Auto-active verification · Object-oriented verification · Verification benchmarks

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—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 predefined, possibly implicit properties such as the absence of invalid pointer dereferences. At the other end of the spectrum are *interactive* approaches to verification—which include tools such as KIV [15]—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. The classification into automatic and interactive is ultimately fuzzy, but it is nonetheless practically useful to assess the level of abstraction at which the user/tool interaction takes place. In the trade-off between expressiveness and scalability, automatic tools tend to prioritize the latter, and interactive tools the former.

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 [33] 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 ("off-line") through *annotations* (such as loop invariants or intermediate lemmas) in the input program. Provided such annotations belong to a higher level of abstraction than the proof details exposed by purely interactive tools, auto-active tools have a chance to be more approachable by non-experts while retaining full support for proving arbitrary properties.

More broadly, the auto-active approach has the potential to better support *incrementality*: proving simple properties would require little annotations and of the

* A preliminary version of this work appeared in the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems in 2015 [50].

** Work mainly done while all the authors where affiliated with ETH Zurich.

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

simple kinds that novice users may be able to provide; 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 mathematical types and theories, 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 problems we use to evaluate AutoProof (Sect. 6) include verification challenges, a realistic container library, and the performance of students on a master's course project. Verification challenges are emerging as the gold standard [28] 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. The container library—called EiffelBase2, and discussed in detail in related work [14]—offers all the features customary in modern language frameworks with realistic implementations; formally verifying its full functional correctness poses challenges that go beyond those of individual benchmark problems. Master's students represent serious non-expert users of AutoProof, who helped us assess to what extent auto-active verification was incremental, and highlighted a few features critical for AutoProof's usability at large.

Previous work of ours, summarized in Sect. 2.2, described the individual techniques available in AutoProof. This paper demonstrates how those techniques together make up a comprehensive verification methodology (Sect. 4), and in addition presents AutoProof's user interface (Sect. 3), describes significant aspects of its design and implementation (Sect. 5), and outlines the results of experiments (Sect. 6) 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://tiny.cc/autoproof`

contains source and binary distributions, detailed usage instructions, a user manual, an interactive tutorial, and the benchmark solutions discussed in Sect. 6.

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 [16] belongs to this category, as it works on Java programs annotated with a variant of JML (the Java Modeling Language [29]). Since it lacks a full-fledged methodology for reasoning about object consistency and framing, using Krakatoa to verify object-oriented idiomatic patterns—such as those we discuss in Sects. 6.1 and 6.2—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 [7] or the more recent OpenJML [38, 11], as well as to the KeY system [5, 1]—which started out as an interactive prover but has provided increasingly more support for automated reasoning. Even when ESC/Java2 was used on industrial-strength case studies (such as the KOA e-voting system [27]), the emphasis was on modeling and correct-by-construction development, and verification was normally applied only to limited parts of the systems. KeY has been successfully applied to a variety of program domains such as information flow analysis [4] and runtime verification [8], with a lesser focus on object-oriented heap-manipulating programs.

By contrast, the Spec# system [2] was the forerunner in a new research direction, also followed by AutoProof, which 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. 6.1) require, however, more flexible methodologies [42] not currently available in Spec#.

Tools, such as VeriFast [24], based on separation logic, provide powerful methodologies through abstractions different from 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. To address this shortcoming, recent research has focused on reducing the amount of low-level annotations by inferring them [39]; we provide a more direct comparison of AutoProof with separation logic provers in related work [14].

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 [32] 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 [45] and Why3 [17], for example, work on functional programming languages—respectively, a subset of Scala and a dialect of ML; VCC [10] works on C programs and supports object invariants, but with an emphasis on memory safety of low-level concurrent code.

AutoProof occupies a sweet spot between automatic and interactive in the wide spectrum of verification tools. The CodeContract static checker (formerly known as Clousot [37]) is a powerful static analyzer for .NET languages that belongs to the former category of fully automatic tools (and hence it is limited to properties expressible in its abstract domains). The KIV environment [15] for Java belongs to the latter category: its full-fledged usage requires explicit user interaction 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 [48]. An important aspect for usability is improving feedback when verification fails; to this end, we introduced heuristics known as "two-step verification" [49] and demonstrated them on algorithmic challenges [46]. We presented the theory behind AutoProof's invariant methodology [42], which includes full support for class invariants, framing, and ghost code. We later added support for model-based specifications and inheritance, and used it to verify the full functional correctness of a realistic container library [14]. The current paper discusses how all these features are available in AutoProof, with a focus on advanced object-oriented verification challenges.

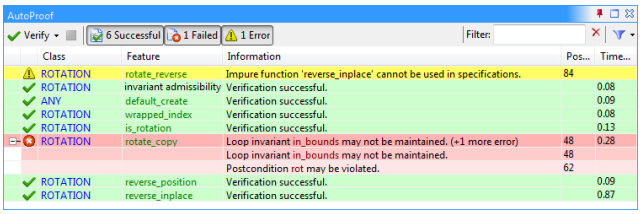

Fig. 1. The AutoProof output panel showing verification results in EVE.

3 Using AutoProof

AutoProof is a static verifier for Eiffel programs which interacts with users according to the auto-active paradigm [33]: 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;[2] routine calls take place in states satisfying the callee's precondition; loops and recursive calls terminate; integer variables do not overflow;[3] there are no dereferences of **`Void`** (Eiffel's equivalent of **`null`** in other languages) objects.

AutoProof's techniques are *sound*:[4] 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).

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://tiny.cc/web-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.

[2] Maintaining invariants is the default, which can be overridden; see Sect. 4.5 for details.

[3] Overflow checking can be disabled to treat integers as mathematical integers.

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

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 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 the 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 (Sect. 3.2) 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.

Verifier's options. 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, on a per-class or per-feature basis, through program annotations that express *verification options*. Using **note** clauses (Eiffel's mechanism for meta-annotations), users can disable generating boilerplate implicit contracts, skip verification of a specific class, disable purity checking of some or all functions, disable termination checking (only verify partial correctness), and define a custom mapping of a class's type to a theory of the underlying reasoning engine. AutoProof's manual includes a complete list of features, options, and examples of usage.

3.2 Two-step verification

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*" [49].

When two-step verification is enabled, each failed verification attempt is transparently followed by a second attempt that uses inlining (of routine bodies) and unrolling (of loop bodies). In other words, the first verification step performs standard modular reasoning—using a called routine's specification or a loop's invariant to reason about its effects within the callee—whereas the second verification step directly uses bodies while ignoring specifications.

The second step is in general unsound (as it uses underapproximations such as loop unrolling[5]), but helps discern whether failed verification is due to real errors or just to insufficiently detailed annotations. For example, consider a routine `sort` that relies on another routine `swap` to rearrange elements in an array. If the verification of `sort` fails using modular reasoning (first step), but succeeds with `swap`'s body inlined within `sort` (second step), it means that `swap`'s specification—precisely its postcondition—is too weak a characterization of `swap`'s effects within `sort` to reason about them.

When two-step verification is enabled, users see the combined output from the two steps in the form of suggestions to improve the program, its annotations, or both. In the example of `sort`, AutoProof suggests strengthening `swap`'s postcondition until modular verification succeeds, or adding a special annotation to `swap` so that AutoProof will always inline it (which is sound in this particular case, because no approximation is involved). In either case, users are aware that verification fails not because there is an error in the implementations of `sort` or `swap` (which would be revealed by inlining), but because more detailed annotations are needed. AutoProof provides similar suggestions for loops about their invariants: if verification of a loop fails in the first step but succeeds with finite loop unrolling, AutoProof deduces that there are no obvious errors in the loop and suggests strengthening the loop invariant to make it inductive.

4 The theory behind AutoProof: specification and verification methodology

AutoProof inputs Eiffel programs, each program a collection of classes, complete with assertions and other kinds of annotations. The Eiffel language is broadly supported, with the exception of a few features that we outline in Sect. 4.1. Expressing concisely complex specifications requires mathematical abstractions, such as sets and sequences, and ghost code, used to record redundant state information necessary for constructing proofs: we describe how AutoProof flexibly supports such constructs in Sects. 4.2, 4.3, and 4.4. Reasoning about realistic object-oriented programs also requires a comprehensive methodology to express dependencies between objects: we give a brief introduction to *Semantic Collaboration*, AutoProof's powerful approach to reasoning about heap-manipulating programs, in Sect. 4.5.

4.1 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

[5] Somewhat similarly to other verification techniques like bounded model checking [9].

types, multiple inheritance, polymorphism—are fully supported [48], and so are imperative and procedural constructs.

Basic annotations. Fig. 2 shows an example of annotated input: an implementation of binary search (problem BINS in Tab. 1) that AutoProof automatically verifies. 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 a loop **`invariant`** (to establish partial correctness) and a **`variant`** (to establish termination). Each specification element consists of clauses, one per line, possibly with *tags* (such as *`sorted`* for the lone precondition clause), which are just names used for identification in error reports.

Quantified expressions in contracts use the **`across`** syntax, which corresponds to universal (**`across`** ... **`all`**) and existential (**`across`** ... **`some`**) (bounded) first-order quantification. For example, the loop invariant clause *`not_left`* expresses the quantification:

$$\forall i\colon 1 \leq i < \texttt{low} \implies \texttt{a.sequence}\,[i] < \texttt{value}\,.$$

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 does not support specification expressions with side effects (for example, a precondition whose evaluation creates an object). It also does not support the semantics of **`once`** routines (somewhat similar to **`static`** in Java and C#), which would require global reasoning, thus breaking modularity. This is not a significant limitation in practice, since **`once`** routines are used only infrequently in Eiffel programs.

Other partially supported features originate in the distinction between machine and mathematical representation of types. Among primitive types, `INTEGER`s are fully supported, and users can choose whether AutoProof should model them as mathematical integers or as machine integers (including overflows). Floating-point `REAL`s are only modeled as infinite-precision mathematical reals, which are different in general from their runtime behavior. `STRING`s are not supported except for single-character operations.

Arrays and lists with simplified interfaces are supported out of the box. Other container types require custom specification; in a closely related effort, we recently developed a fully verified full-fledged data structure library [14], which we describe in Sect. 6.2.

Agents (function objects) are partially supported, with some restrictions in their specifications [48]. The semantics of native **`external`** routines is reduced to their specification. We designed [48] a translation for exceptions based on the latest draft of the Eiffel language standard, but AutoProof does not support it yet since the Eiffel compiler still only implements the obsolete syntax for exceptions. However, exceptions have very limited usage in Eiffel programs compared to other object-oriented languages; hence, this limitation has little practical impact.

AutoProof is currently limited to *sequential* code, which excludes support for threaded libraries and the SCOOP concurrency mechanism [52].

4.2 Logic classes and specification library

Functional specifications should abstract from implementation details to capture the gist of what an algorithm or program does. To this end, they are naturally expressed in terms of mathematical structures such as sets and sequences, which uphold abstract specifications and have a direct mapping to the underlying theorem prover's logic. For example, the specification of binary search in Fig. 2 refers to the content of the input array `a` through its attribute `sequence`, which represents the mathematical sequence of integer values stored in `a`'s cells; this way, binary search's specification abstracts away from the detail that the implementation operates on an array and remains applicable to any data structure storing a sequence of integer values.

To support abstract and concise, yet expressive, specifications, AutoProof provides *logic classes*: immutable wrapper classes that encapsulate mathematical structures, whose instances and operations are mapped directly into theories of the underlying reasoning engine (as we describe in Sect. 5.2) instead of being treated as regular Eiffel classes.

AutoProof comes with a standard library of logic classes called MML (mathematical model library), which provides relations, sets, sequences, bags (multisets), and maps, and common operations on them. In Fig. 2, attribute `a.sequence` has type `MML_SEQUENCE [INTEGER]`, which is MML's class for mathematical sequences.

An advantage of providing mathematical types by means of a library of classes instead of natively—as most other auto-active verifiers do—is that a library is easily *extensible* with new structures and new operations. In Sect. 5.2, we discuss how this is possible in AutoProof. While extending MML is normally not necessary, and certainly not required, the extension mechanisms provide great flexibility to the most advanced users of AutoProof.

4.3 Model-based contracts

AutoProof supports model-based contracts: an approach to writing interface specifications that helps maintain a consistent level of abstraction among features of a class, so as to provide clients with all the information necessary

```
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.

to reason about the observable behavior of the class instances without revealing implementation details. Each class declares a **model** clause, which designates a subset of the class's attributes as the class *model*. The model should characterize the publicly observable *abstract state* of the class, which is the only information clients of the class can rely on. For example, the model of a stack is the sequence of values stored in the stack, listed, say, from bottom to top; correspondingly, class STACK in Fig. 3 declares attribute sequence of class MML_SEQUENCE (from the MML library described in Sect. 4.2) as its **model**. Implementation details, such as how the stack internally stores elements in a list, are invisible to clients, and hence not part of the model. Specifications of routines refer to the model, such as push's postcondition: pushing an element v onto the stack extends sequence by appending v to the tail.

By using model-based contracts, we can also define the *completeness* of interface specifications (see our previous work for a formal definition [40]). In Eiffel, it is customary to follow the command/query separation design principle, whereby class features are either commands or queries: a command is a state-modifying routine that returns no value; a query is a pure function that returns a value without changing the object state. Given model-based specifications of a class, a command's postcondition is complete if it uniquely defines the effect of the command on the model; a query's postcondition is complete if it defines the returned result as a function of the model; the class model is complete if it permits expressing complete specifications of all public routines in the class such that different abstract states are distinguishable by public routine calls. This means that a set is not a complete model for STACK in Fig. 3, because the precise result of item cannot be defined as a function of a set (where the order of elements is immaterial); conversely, a sequence would not be a complete model for a class SET because no features in the interface of SET can discriminate different element orderings.

AutoProof currently does not support mechanized completeness proofs based on model-based contracts; however, we found that even reasoning informally about completeness helps provide clear guidelines for writing interface specifications and substantiates the notion of full functional correctness.

4.4 Ghost code

Ghost code identifies program elements—variables, instructions, and routines—that are only used in specifications and verification, and erased upon compilation (hence, not present in the executable). In the auto-active paradigm, ghost code provides advanced support for abstraction and for indirectly interacting with the verifier

```
class STACK
model sequence

  list: LIST [G]

  sequence: ghost MML_SEQUENCE [G]

  item: G
    require
      not is_empty
    do
      Result := list.item (count)
    ensure
      Result = sequence.last
    end

  count: INTEGER
    do
      Result := list.count
    ensure
      Result = sequence.count
    end

  push (v: G)
    modify model [sequence] Current
    do
      list.extend_back (v)
    ensure
      sequence = old sequence & v
    end
  ...
invariant
  list ≠ Void
  owns = {list}
  sequence = list.sequence
end
```

Fig. 3. Excerpt from class STACK, which uses a LIST as its internal representation. The specification of STACK uses the model class MML_SEQUENCE to refer to the mathematical sequence of stack elements.

in a way that goes beyond assertions on the concrete program state.

The model attributes of a class are often (but not always) ghost, such as sequence in Fig. 3. Indeed, the sequence of elements represents redundant information, since it corresponds to the content of list as specified in the last clause of the class invariant; hence, it is only used to support abstract specifications and effective reasoning, but it is not compiled so as not to introduce unnecessary runtime overhead.

Another usage of ghost code is to represent lemmas: a ghost routine lm with precondition P and postcondition Q represents the formula $P \Longrightarrow Q$; calling lm makes the formula available to the prover at the call site; the body of lm builds a proof of the formula following the code-as-proof paradigm [25].

More generally, users can mark variables and other program elements as ghost. Since Eiffel lacks native support for ghost code, AutoProof uses **note** status: **ghost** to this effect. In Fig. 3, as well as in the other code snippets in the paper, we simplify the syntax for readability, and use **ghost** and other AutoProof-specific annotations as if they were native keywords.

4.5 Heap-manipulating programs: framing and invariants

Reasoning effectively about heap-manipulating programs in general, and object-oriented ones in particular, requires flexible support for framing and object-consistency specification.

4.5.1 Framing

Framing is the fundamental part of the specification of a command that describes which elements in the heap the command may modify. A dual notion applies to queries (pure functions), whose specification must describe which elements in the heap the query may read. AutoProof provides support for framing specifications in the style of dynamic frames [26]. A command whose execution may modify objects in set declares its frame specification using a clause **modify** (set). Since Eiffel doesn't natively support modify clauses, AutoProof introduces a dummy feature named **modify**, which can be used in a routine's precondition to specify the frame, as it is done in various places in Fig. 4.

Model attributes support a special kind of frame specifications: a routine annotated with the clause

modify model [$\mathtt{m}_1$,...,$\mathtt{m}_n$] s

may only modify attributes $\mathtt{m}_1$,...,$\mathtt{m}_n$ in the abstract state of s (the model), with no direct restrictions on modifying the concrete state of s. Routine push in Fig. 3 uses this construct to specify that pushing an element onto the stack modifies the model attribute sequence of the **Current** instance (denotes by this in other languages); this stipulates that it may modify **Current**.list, but it should not modify any model attribute other than **Current**.sequence.[6] This construct enables fine-grained, yet abstract, frame specifications, in a style similar to data groups [36].

4.5.2 Invariants: object consistency

Class invariants are a natural way to define when an object is in a consistent state. However, specifying con-

[6] Even though class STACK does not explicitly define any other model attributes, such attributes might be added in descendant classes; in addition, the invariant methodology described below equips each class with implicit model attributes owns, subjects, and observers.

```
class SUBJECT
  value: INTEGER
  subscribers: LIST [OBSERVER]

  update (v: INTEGER)
    require
      across observers as o all o.wrapped end
    modify (Current, observers)
    do
      unwrap_all (observers)
      value := v
      across subscribers as o do o.notify end
      wrap_all (observers)
    ensure
      across observers as o all o.wrapped end
      observers = old observers
    end

  register (o: OBSERVER) -- Internal
    require
      not subscribers.has (o)
      wrapped
      o.open
    modify (Current)
    do
      unwrap
      subscribers.add (o)
      observers := observers & o
      wrap
    ensure
      subscribers.has (o)
      wrapped
    end

invariant
  observers = subscribers.range
  owns = { subscribers } and subjects = {}
end
```

```
class OBSERVER
  subject: SUBJECT
  cache: INTEGER

  make (s: SUBJECT) -- Constructor
    require
    modify (Current, s)
    do
      subject := s
      s.register (Current)
      cache := s.value
      subjects := { s }
    ensure
      subject = s
    end

  notify -- Internal
    require
      open
      subjects = { subject }
      subject.observers.has (Current)
      observers = {}
      owns = {}
    modify (Current)
    do
      cache := subject.value
    ensure
      inv
    end

invariant
  cache = subject.value
  subjects = { subject }
  subject.observers.has (Current)
  observers = {}
  owns = {}
end
```

Fig. 4. The *observer pattern* in AutoProof using semantic collaborations. See [42] for a detailed explanation.

sistency may become tricky when multiple objects that are independently accessible depend on each other: if the invariant of some object x depends on the state of another object y, any change to y may invalidate x indirectly. This may break modularity, because x may have no direct control of the calls to y that modifies its state. To handle such object dependencies, AutoProof features two integrated mechanisms: a standard *ownership* scheme [34], combined with a novel methodology called *semantic collaboration*, which we introduced in a previous work [42].

Common to the two methodologies is the notion of open/closed objects: an object is closed when it is in a stable, consistent state where its invariant holds; conversely, it is open when it is in a transient state where its invariant may not hold. AutoProof equips every object with the implicit ghost Boolean attribute `closed`, which holds iff the object is closed. Attribute `closed` is manipulated indirectly by calling built-in routines `unwrap` and `wrap`. Informally, a "canonical" modification of an object `o` involves following a three-part protocol: signal that `o` is being modified (and hence its invariant cannot be relied upon) by calling `o.unwrap`; perform the actual

modification; check that `o` is back in a consistent state (and hence its invariant holds) by calling `o.wrap`.

Ownership is suitable to specify object dependencies that are *hierarchical*. This is the case for the dependency between instances of `STACK` in Fig. 3 and the object attached to their `list` attribute, since the latter is part of the stack's internal representation. Following a semantic treatment similar to that available in VCC [10], AutoProof introduces an implicit ghost attribute `owns`, which is used to specify the set of objects that are "owned" by the current instance, and hence are part of **`Current`**'s internal representation (`owns` = {`list`} in `STACK`'s invariant). AutoProof checks the property that any object cannot be unwrapped, and thus modified, as long as it has a closed owner, thereby effectively ensuring that all modifications to an owned object (the list, in the example) go through its owner (the stack).

Semantic collaboration is suitable to specify object dependencies that are *not* hierarchical but "peer-to-peer". Two objects x and y form a collaborative (non-hierarchical) structure when they collaborate to achieve a common goal while retaining access independence.

The observer pattern is a paradigmatic example of collaborative structure, where an arbitrary number of `OBSERVER` objects (called "subscribers") monitor the public state of a single instance of class `SUBJECT`: see a fully annotated implementation in Fig. 4. Each subscriber caches a copy of the subject's attribute `value` in its local attribute `cache`. The fundamental consistency property is that the value of `cache` remains a faithful copy of `value` in the subject. This property is specified by the invariant clause `cache` = `subject.value` in class `OBSERVER`, which introduces a dependency on another object's state; since subscribers are peers, and we cannot realistically assume that one of them has exclusive control over the subject, the dependency between `OBSERVER` and `SUBJECT` objects is not hierarchical. Consistency is instead enforced by explicit collaboration: the subject maintains a list of `subscribers`, updated whenever a new subscriber registers itself by calling `register` on the subject. Whenever calls to routine `update` change `value`, the subject takes care of explicitly notifying all registered subscribers (the **`across`** loop in `update`'s body calls `notify` on every object `o` in `subscribers`).

To specify collaboration patterns, AutoProof equips objects with implicit ghost attributes `subjects` and `observers`.[7] Given an object `o`, `o.subjects` stores the set of objects on which `o`'s invariant may depend, and `o.observers` stores the set of objects whose invariant may depend on `o`. (In the example, the invariant clause `observers` = `subscribers.range` specifies that the observers' consistency depends on the state of the subject.) AutoProof checks that every update of `o` preserves the validity of its `observers` (in the example, that routine `update` notifies the observers). AutoProof also checks that the "subject" and "observer" relations are each other's inverse (in the example, the invariant clause `subject.observers.has` (**`Current`**)), so that the subject cannot cheat by kicking a subscriber out of its `observers` set (and hence satisfying the specification by dumping contractual obligations).

[7] While the names are inspired by the observer pattern, they are applicable also to many other collaboration patterns, as we extensively demonstrated in related work [42,41].

4.5.3 Default annotations

Defaults reduce the amount of required manual annotations in many practical cases. AutoProof offers defaults for the following elements:

Preconditions and postconditions: every public command `m` requires and ensures that `m`'s arguments, **`Current`**, and **`Current`**'s `observers` are `wrapped` (closed and not owned).

Modify clauses: commands may modify **`Current`**; queries may modify nothing.

Invariants: Built-in implicit attributes `owns`, `subjects`, and `observers` are empty if they are not mentioned in programmer-written invariant clauses.

Wrapping: public commands call **`Current`**`.unwrap` as first instruction in their body, and **`Current`**`.wrap` as last instruction.

Implicit ghost attributes: if a ghost attribute a is mentioned in an invariant clause of the form a = `expr`, every `wrap` of objects with a is preceded by the implicit assignment a:= `expr`.[8]

AutoProof's defaults are modeled after a programming style where clients always operate on consistent objects, and objects have to ensure consistency of themselves and all their observers upon terminating the execution of their public routines. Although this style promotes encapsulation and is convenient for clients, the defaults are optional suggestions that can be overridden by providing explicit annotations. The combination of flexibility and useful defaults is instrumental in making AutoProof usable on complex examples of realistic object-oriented programs.

5 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# [2] and since then adopted by numerous other tools, AutoProof does not generate VCs directly but

[8] This default is inspired by VCC's static owns [10].

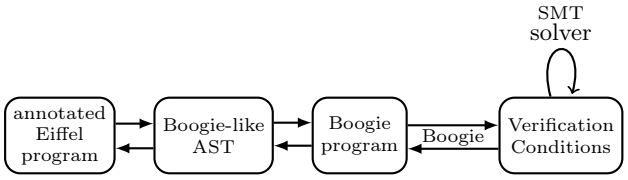

Fig. 5. Workflow of AutoProof with Boogie back-end.

translates Eiffel programs into Boogie programs [31] 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.

As outlined in Fig. 5, 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, it transcribes the AST into a textual Boogie program.

The rest of this section outlines the overall mapping from Eiffel to Boogie (Sect. 5.1) as well as the special mapping of logic classes (Sect. 5.2), and then focuses on describing how AutoProof's architecture (Sect. 5.3) and implementation features make for a flexible and customizable translation process. We focus on discussing the challenges tackled when developing AutoProof and the advantages of our implemented solutions.

5.1 Eiffel to Boogie encoding

We briefly recall the core features of how Eiffel is encoded in Boogie; see our previous work [48] for details.

Since Boogie is a simple procedural language without explicit support for object-oriented features or any form of dynamic memory, a key feature of the encoding is a *memory model*, mapping Eiffel's heap, references, and objects. The `Heap` is a Boogie global variable

```
var Heap: ⟨α⟩ [ref, Field α] α
```

which maps pairs of reference (type **`ref`**) and field name (type `Field`) into values.

The *heap* type is polymorphic with respect to the value type α, which can be any of the available types—including references **`ref`** and Boogie primitive types such as **`int`** and **`bool`**. In the encoding, Eiffel *reference types* become **`ref`** in Boogie, whereas Eiffel *primitive types* become Boogie primitive types with axiomatic constraints to ensure consistent representation. For example, Eiffel variables of type `INTEGER` map to Boogie variables of type **`int`**; the latter, however, corresponds to mathematical integers, whereas the former has 32-bit precision. To bridge the semantic gap, constraints such as $-2^{31} \leq \texttt{iv} \leq 2^{31} - 1$, where `iv` is a variable of type **`int`**, are used in the Boogie encoding to check for absence of overflows in operations that manipulate Eiffel integers.

An Eiffel *class declaration* of the form

```
class C
inherit B
feature a: A
...
```

introduces a class `C`, inheriting from another class `B`, that includes an attribute `a` of type `A`. The Boogie encoding of `C` comprises an uninterpreted type identifier **`const unique`** `C: Type` related, in an axiom, to `B`'s type identifier by the "subtype of" partial order relation `C <: B`. The axioms about classes are built around this relation so as to support polymorphism.

`C`'s *attribute* `a` determines a name `C.a` declared as **`const`** `C.a: Field` **`ref`**, which is used to access the object attached to `a` in the heap. For example, the Eiffel assignment `c.a := x` (where `c` is a reference of class `C` and `a` is one of `C`'s attributes) is expressed in Boogie as

```
Heap := Heap[c, C.a := x]
```

which updates to `x` the mapping of the pair `(c, C.a)` in global variable `Heap`, while leaving the rest of the mapping unchanged.

Eiffel routines translate to Boogie **`procedure`**s: Eiffel procedures (not returning any value) update the heap according to their modify clauses; Eiffel functions are assumed pure by default, in which case an Eiffel function `f` of class `C` translates to both a Boogie **`procedure`** `C.f` and a Boogie **`function`** `fun.C.f`. The former's postcondition includes a check that `C.f` returns values consistent with `fun.C.f`'s definition. Then, `C.f` is used to check `f`'s correctness against its specification and to reason about usages of `f` in imperative code; `fun.C.f` encodes `f`'s values when they are used in specification elements.

The translation of *control-flow* structures (conditionals and loops) uses the corresponding Boogie structures whenever possible.

5.2 Implementation of logic classes

Logic classes (Sect. 4.2) are a mechanism for defining specification constructs with custom semantics; AutoProof handles their translation into Boogie differently from regular Eiffel classes, using a mechanism based on the work of Darvas and Müller [12] for JML. In AutoProof, each logic class declares, by means of a **`maps_to`** clause, a *Boogie type* that will represent the class type in Boogie. The constructors and functions[9] of logic classes

[9] Since they are immutable, logic classes do not include state-modifying commands.

```
class MML_SEQUENCE [G]
  inherit ITERABLE [G]
  maps_to Seq
  theory "sequence.bpl"
  typed_sets Seq#Range

  count: INTEGER
    maps_to: Seq#Length

  last: G
    require count >0
    ...

  range: MML_SET [G]
    ...

  extended (x: G): MML_SEQUENCE [G]
    alias "&"
    ...

  to_bag: MML_BAG [G]
    ...

  shorter_equal (s: MML_SEQUENCE [G]): BOOLEAN
    alias "≤ "
    ...

  new_cursor: CURSOR [G]
    maps_to Seq#Range
end
```

Fig. 6. Model class `MML_SEQUENCE` equipped with AutoProof *logic class* annotations. Features with no **`maps_to`** clause use the default naming scheme: for example, `last` maps to `Seq#Last`.

are also directly connected to Boogie functions using a **`maps_to`** clause—in the absence of such a clause, AutoProof looks for a Boogie function with a name derived from the Eiffel feature name. Fig. 6 shows how the `MML_SEQUENCE` logic class uses the **`maps_to`** clause.

Using logic classes requires Boogie theories that encode the classes' features together with useful lemmas and other axiomatic definitions. To this end, AutoProof supports the **`theory`** clause, which specifies a Boogie file that is to be included with every translation of an Eiffel program that uses a specific logic class. Fig. 7 shows an excerpt of the Boogie theory `sequence.bpl`, which encodes the operations of the logic class `MML_SEQUENCE`.

AutoProof also implements a number of advanced features, which aim at making custom logic classes as expressive and convenient as built-in mathematical types. We describe the most significant ones in the remainder of this section.

Quantifiers. Eiffel's **`across`** syntax

```
across s as x all B(x) end
across s as x some B(x) end
```

supports quantification over any object structure `s` that inherits from class `ITERABLE` and implements a feature `new_cursor`. If `s` is a logic class whose `new_cursor` feature maps to a set-valued function, AutoProof translates the **`across`** expressions into first-order quantification over the domain defined by the Boogie translation of `new_cursor`. For example, **`across`** expressions over objects of class `MML_SEQUENCE` translate to first-order quantification over the range of elements in a sequence, as per Fig. 6.

Ordering. If a logic class `L` includes a feature with **`alias`** "$\leq$" (such as `shorter_equal` in Fig. 6), AutoProof assumes that the feature defines a well-founded order on the values of `L`. Hence, loop variants can use expression of type `L` to prove termination.

Element types are a technique [31] to map several generic reference types of a source language to a single Boogie type, while encoding more precise type information using automatically generated predicates. In AutoProof, this technique helps encode logic classes, which often represent immutable collections of elements of generic type—such as class `MML_SEQUENCE [G]` with respect to the generic type parameter `G`. Whenever a logic class is used in annotations, AutoProof has to record the information about how its generic type parameter is instantiated with a concrete type. For example, an Eiffel declaration `s: MML_SEQUENCE [PERSON]` entails that all elements of `s` are of type `PERSON` (or its subtypes via polymorphism).

This is done concisely using element types: AutoProof's **`typed_sets`** clause provides a way to associate a set-valued Boogie function `ts` to each generic parameter `G` of a logic class, such that `ts` defines all objects of type `G` collected in an instance of the logic class. For example, the declaration **`typed_sets`** `Seq#Range` in Fig. 6 makes AutoProof generate, for the Eiffel declaration `s: MML_SEQUENCE [PERSON]`, a Boogie axiom stating that the Eiffel type of every object in the range of `s` is a subtype of `PERSON`.

Type constraints. Logic classes may imposes constraints on the values of their instances that have to hold whenever a value of the class is chosen nondeterministically. AutoProof supports such constraints using `where` clauses. For example, logic class `MML_BAG` includes a `where` clause to state that the multiplicity (number of occurrences) of every elements in a bag is a nonnegative number.

To our knowledge, AutoProof is the only auto-active verifier with support for extensible mathematical types that enjoy the same level of language integration and underlying prover support as built-in types in other verifiers.

```
// Sequence type
type Seq T;

// Sequence length
function Seq#Length<T>(Seq T): int;

// Element at a given index
function Seq#Item<T>(Seq T, int): T;

// Last element
function Seq#Last<T>(q: Seq T): T
{ Seq#Item(q, Seq#Length(q)) }

// Set of values
function Seq#Range<T>(Seq T): Set T;
axiom (∀<T> q: Seq T, o: T •Seq#Has(q, o) ⟺Seq#Range(q)[o]);

// Sequence extended with x at the end
function Seq#Extended<T>(s: Seq T, val: T): Seq T;
axiom (∀<T> s: Seq T, v: T •
  Seq#Length(Seq#Extended(s,v)) =1 + Seq#Length(s));
axiom (∀<T> s: Seq T, i: int, v: T •
  (i =Seq#Length(s) + 1 ⟹Seq#Item(Seq#Extended(s,v), i) =v) ∧
  (i ≤Seq#Length(s) ⟹Seq#Item(Seq#Extended(s,v), i) =Seq#Item(s, i)));

// Sequence converted to a bag
function Seq#ToBag<T>(Seq T): Bag T;
axiom (∀<T> •Seq#ToBag(Seq#Empty(): Seq T) =Bag#Empty(): Bag T);
axiom (∀<T> s: Seq T, v: T •
  Seq#ToBag(Seq#Extended(s, v)) =Bag#Extended(Seq#ToBag(s), v));

// Is |q0| ≤|q1|?
function Seq#LessEqual<T>(q0: Seq T, q1: Seq T): bool
{ Seq#Length(q0) ≤Seq#Length(q1) }
```

Fig. 7. An excerpt from `sequence.bpl`, which contains a manual Boogie encoding of `MML_SEQUENCE`.

5.3 Extensible architecture

The architecture of AutoProof deploys widely used design patterns to facilitate the reuse and extension of its functionality.

5.3.1 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.

5.3.2 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 an AST contains all the information necessary for verification. The transcription of an AST into a concrete Boogie program is implemented by a *visitor* [21] of the AST. Modifying AutoProof in response to changes in Boogie's syntax would only require to modify the visitor.

5.3.3 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 num-

ber 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 [21]; 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.

5.3.4 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).

Modular reasoning, in turn, permits a modular translation from Eiffel to Boogie, which only includes code elements necessary for the current verification targets. 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 has not 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.

5.3.5 Traceability of results

The auto-active paradigm is based on interacting with users at the high level of the source language as much as possible; 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, 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 [42]).

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.

5.3.6 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; the results are fed back to users when verification of the whole input is 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.

6 Evaluation

The proof of AutoProof is in the pudding. To demonstrate its flexibility in supporting verification challenges,

we report the results of a three-part evaluation. In Sect. 6.1, we use a collection of 30 benchmark verification problems to demonstrate AutoProof's capabilities on challenges that are recognized as the yardstick to evaluate full functional verification outside of the laboratory. In Sect. 6.2, we summarize our recent effort of verifying the full functional correctness of a container library with realistic implementations; its size and complexity substantiate the claim that AutoProof can deal with realistic general-purpose code beyond isolated exercises. In Sect. 6.3, we discuss the performance of master's students using AutoProof in developing a project for a software verification course; the usability by serious non-expert users gives some evidence to the claim that auto-active verifiers provide incrementality and identifies hurdles to improving usability.

6.1 Benchmarks

We give capsule descriptions of benchmark problems that we verified using the latest version of AutoProof; the complete solutions are available at

`http://tiny.cc/autoproof-repo`

through AutoProof's Web interface.

6.1.1 Benchmarks description

Our selection of problems is largely based on the verification challenges put forward during several scientific forums, namely the SAVCBS workshops [43], and various verification competitions [28,6,18,22,23] and benchmarks [51]. 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 [42] for our solutions to problems 13–20). The table is partitioned into three groups: the first group (1–11) includes mainly *algorithmic* problems; the second group (13–20) includes object-oriented design challenges that require complex *invariant* and *framing* methodologies; the third group (21–30) 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 invariant depends on objects not directly accessible from the node in the physical heap.

6.1.2 Verified solutions with AutoProof

Tab. 2 displays data about the verified solutions to the problems of Sect. 6.1; 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 S/C specification to code ratio (measured in tokens)[10]; 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 of 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.

6.1.3 Discussion

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 verification 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 are 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

[10] 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	DESCRIPTION	FROM
1	Arithmetic (ARITH)	Build arithmetic operations based on the increment operation	[51]
2	Binary search (BINS)	Binary search on a sorted array (iterative and recursive version)	[51]
3	Sum & max (S&M)	Sum and maximum of an integer array	[28]
4	Search a list (SEARCH)	Find the index of the first zero element in a linked list of integers	[28]
5	Two-way max (2-MAX)	Find the maximum element in an array by searching at both ends	[6]
6	Two-way sort (2-SORT)	Sort a Boolean array in linear time using swaps at both ends	[18]
7	Relaxed prefix (RPRE)	Determine if one sequence is a prefix of another possibly modulo an extra character	[23]
8	Dutch flag (DUTCH)	Partition an array in three different regions (specific and general verions)	[13]
9	LCP (LCP)	Longest common prefix starting at given positions x and y in an array	[22]
10	Rotation (ROT)	Circularly shift a list by k positions (multiple algorithms)	[19]
11	Sorting (SORT)	Sorting of integer arrays (multiple algorithms)	
12	Concurrent GCD (GCD)	Nondeterministic Euclid's algorithm with termination guarantees	[23]
13	Iterator (ITER)	Multiple iterators over a collection are invalidated when the content changes	[43, '06]
14	Subject/observer (S/O)	Design pattern: multiple observers cache the content of a subject object	[43, '07]
15	Composite (CMP)	Design pattern: a tree with consistency between parent and children nodes	[43, '08]
16	Master clock (MC)	A number of slave clocks are loosely synchronized to a master	[3]
17	Marriage (MAR)	Person and spouse objects with co-dependent invariants	[35]
18	Doubly-linked list (DLL)	Linked list whose nodes have links to left and right neighbors	[35]
19	Dancing links (DANCE)	Doubly-linked list where nodes can be un-removed	[23]
20	PIP (PIP)	Graph structure with cycles where each node links to at most one parent	[44]
21	Closures (CLOSE)	Various applications of function objects	[30]
22	Strategy (STRAT)	Design pattern: a program's behavior is selected at runtime	[30]
23	Command (CMD)	Design pattern: encapsulate complete information to execute a command	[30]
24	Map ADT (MAP)	Generic map ADT with layered data	[51]
25	Linked queue (QUEUE)	Queue implemented using a linked list	[51]
26	Tree maximum (TMAX)	Find the maximum value in nodes of a binary tree	[6]
27	Ring buffer (BUFF)	A bounded queue implemented using a circular array	[18]
28	Hash set (HSET)	A hash set with mutable elements	
29	Board game 1 (GAME1)	A simple board game application: players throw dice and move on a board	
30	Board game 2 (GAME2)	A more complex board game application: different board-square types	

Table 1. Descriptions of benchmark problems.

#	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	RPRE	1	1	0	39	25	0	11	0	12	2	0	0	2.9	547	1355	2.8
8	DUTCH	1	4	1	72	75	13	22	4	21	0	15	0	2.6	1447	1355	4.1
9	LCP	2	2	0	40	28	4	7	0	6	2	9	0	1.0	1359	1355	4.2
10	ROT	1	3	3	51	74	14	10	3	17	2	28	0	2.6	1138	1355	4.1
11	SORT	1	9	6	177	219	31	38	9	56	5	80	0	2.6	2302	1355	5.8
12	GCD	1	1	2	31	33	3	2	0	4	1	23	0	2.9	495	1355	3.3
13	ITER	3	8	0	88	69	15	26	6	0	0	11	11	1.4	1461	1355	8.9
14	S/O	3	6	0	71	56	10	14	4	3	0	15	10	1.4	1156	1355	4.4
15	CMP	2	5	3	54	125	19	18	5	0	2	72	9	4.3	1327	1355	7.5
16	MC	3	7	0	63	61	9	14	5	0	0	26	7	1.8	956	579	3.7
17	MAR	2	5	0	45	50	12	11	3	0	0	19	5	2.3	755	579	3.3
18	DLL	2	8	0	69	76	12	14	4	0	0	39	7	2.0	891	579	4.4
19	DANCE	2	9	2	98	137	22	24	11	0	0	73	7	3.0	1272	579	4.9
20	PIP	2	5	1	54	111	23	18	6	0	1	56	7	3.9	988	1355	5.8
21	CLOSE	9	18	0	145	106	40	31	8	0	0	22	5	0.8	2418	688	5.7
22	STRAT	4	4	0	43	5	0	4	0	0	0	1	0	0.2	868	579	3.3
23	CMD	6	8	0	77	32	4	14	2	0	0	10	5	0.7	1334	579	3.3
24	MAP	1	8	0	78	67	6	29	2	6	4	15	5	2.3	1259	1355	4.1
25	QUEUE	4	13	1	121	101	11	26	1	0	0	48	15	1.5	2360	1355	7.4
26	TMAX	1	3	0	31	43	3	12	2	0	2	19	5	2.1	460	1355	3.2
27	BUFF	1	9	0	66	54	8	19	4	0	0	12	11	1.1	1256	1355	4.4
28	HSET	5	14	5	146	341	45	39	10	20	2	197	28	3.7	3546	1355	13.7
29	GAME1	4	8	0	165	93	16	13	4	31	3	10	16	1.2	4044	1355	26.6
30	GAME2	8	18	0	307	173	25	27	11	48	3	29	30	1.4	7037	1355	54.2
	total	76	195	26	2430	2360	379	492	109	262	47	880	191	1.9	45403	1355	212.6

Table 2. Verification of benchmark problems with AutoProof.

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 [32] and Why3 [17], 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 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 (see Sect. 6.3), 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 only 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. 4.1), floating point numbers are translated as infinite-precision reals; precise reasoning requires manually specifying properties of floating point operations, which is impractical in all but the simplest cases. 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.

6.2 Full verification of a container library

One of the original goals of AutoProof's research—and a driving force behind designing its features—was verifying a fully specified realistic data structure library. We achieved the goal with the complete verification of full functional correctness of the container library EiffelBase2. The library and the verification effort are described in detail in our previous work [41]. In this section, we give an idea of the challenges of the process and a summary how AutoProof supported it.

EiffelBase2 is a *realistic* implementation of *generic* reusable data-structure components. The codbase has substantial size: verified EiffelBase2 consists of 46 classes offering an API with 135 public routines; its implementation has over 8000 lines of code and annotations in 79 abstract and 378 concrete routines. The object-oriented design and realistic implementation compound the complexity of verification: the API is rich in features, as one would expect from similar libraries such as Java's `java.util` containers, yet abstract; (multiple) inheritance is used extensively to relate abstractions and concrete implementations; and the algorithmic implementations do not trade off efficiency for verifiability.

EiffelBase2's overall annotation overhead is 1.4 lines of annotations per line of executable code, or 2.7 tokens of annotation per token of executable code. The overhead is not uniform across classes: abstract classes (such as those defining the common interface of lists) tend to include a lot of annotations which are then amortized over multiple implementations of the same abstract specification. This overhead compares favorably to the state of the art in full functional verification of heap-based data structure implementations (see [41] for details).

On the same hardware used in the benchmark problems (Sect. 6.1), AutoProof takes under 8 minutes to verify the whole EiffelBase2. The distribution of times per routine shows that AutoProof's performance is stable and *predictable*: over 99% of the routines verify in under 10 seconds; over 89% in under 1 second; the most complex routine verifies in under 12 seconds. These uniform, short verification times are a direct result of AutoProof's flexible approach to verification, and specifically of our effort to provide an effective Boogie encoding.

6.3 AutoProof in the classroom

Using AutoProof to verify benchmarks and a full-fledged container library demonstrate its flexibility and performance when used by experts—in particular, by its own developers—but fails to account for its usability by *serious non-experts* [33]. To draw an all-around picture of how AutoProof supports incremental usage by non-experts, we give an account of our experience in using it to support teaching of a master's level course on software verification that we organized at ETH Zurich during the fall semester 2014.[11] In a related publication [20], we reflect in greater detail on AutoProof's usability in the classroom and to verify pre-existing code.

Our "Software Verification" course introduced students to a variety of verification techniques (from abstract interpretation to model checking, from program logics to testing), while emphasizing their unifying traits based on the foundational concepts provided by axiomatic semantics. The course combined lectures, which illus-

[11] See the course's homepage at `http://se.inf.ethz.ch/courses/2014b_fall/sv/`.

trate theory, and exercise sessions and project work, which demonstrate verification in practice.

After a one-hour tutorial introduction to AutoProof, students tackled exercises—during the exercise sessions, tutored by the course's TA—and a project—to be developed off-line in groups of up to three students.

Exercises. Students used the Web interface of AutoProof (Sect. 3.1) to carry out guided tasks of increasing difficulty:

Counter: given an implementation of a wrap-around counter, provide precondition and postcondition of its `increment` routine, so that the whole class verifies.

Hoare triples: encode each of six Hoare triples as a routines of a given stub class and use AutoProof to verify them.

Maximum: given an implementation of two-way array maximum (problem 2-MAX in Tab. 1), provide precondition and quantified loop invariant, so that its given postcondition verifies.

Sum & max: given an implementation of sum & maximum of an array (problem S&M in Tab. 1), provide loop invariant and postcondition that verify and are as detailed as possible.

LCP: given an implementation of the least common prefix routine (problem LCP in Tab. 1) and a series of test cases that call the routine on concrete inputs and check the output, provide pre- and postcondition of the routine so that the test cases verify.

Project. The project consisted in implementing and verifying a simple array-based list data structure class, equipped with basic features to manipulate the list and with a routine to `sort` the list's content. The students received a skeleton Eiffel class with the signatures of the routines they were required to implement, annotate, and verify; for simplicity, the list was limited to storing integer values. Routine `sort` was implemented as a wrapper that calls two sorting algorithms according to how many and what kinds of elements are in the list: if the list contains "many" elements (more than a given constant N) ranging over values between $-V$ and $+V$, `sort` calls a bucket sort implementation; otherwise, it calls a quick sort implementation.

The students developed the project in groups of 1–3 persons over a period of 6 weeks, during which they could get the help of the TA and of the last author of this paper to clarify the behavior and functionality of AutoProof. Since the project involved developing implementation and specification given informal descriptions, the students had a chance to target different trade-offs between sophistication of their implementations, completeness of their specifications, and difficulty of verification.

Results. The exercises gradually introduced the students to the features of AutoProof. In particular, they made them familiar with the way modular verifiers reason about program correctness: the notion that the effects of a call to a routine r are given solely by r's specification independent of its implementation—and, consequently, if r has a weak specification it becomes practically impossible to prove anything about its effects even if its implementation is correct.

All students were able to follow through the exercises, learning the ropes of auto-active verification. Nonetheless, things proved more challenging when they switched over to developing the project. All of the 9 student groups managed to fully implement, specify, and verify the list's basic API features; 6 groups managed to fully and correctly specify the sorting routines, while 3 groups inadvertently introduced inconsistencies in the specifications; and only 5 groups managed to complete a proof of the full specification of the sorting routines including, crucially, the property that the output is a permutation of the input.

These results are a somewhat mixed bag. On the one hand, it is encouraging that students without any previous experience of using AutoProof or similar auto-active tools were able to carry out significant verification tasks including, in the majority of cases, full functional correctness proofs. On the other hand, there remains a chasm in the learning curve of AutoProof, which our students encountered as they progressed toward proving the more complex properties of the sorting algorithms. Initially, they could continue using AutoProof as a black box (as they used it in the exercises); but, at some point, they had to come to terms with the fact that AutoProof may require *redundant* information, in the form of intermediate annotations and lemmas, to get to a successful proof. Understanding when and how this extra information is required is something that can only be learned by trial and error, developing an intuition of how AutoProof works internally.

A related issue is what happened with the 3 groups who had inconsistent specifications and, consequently, could vacuously verify properties with little apparent effort. We, the experts, have learned to be suspicious of complex proofs that go through with little effort and have developed a few simple practices to check for possible inconsistencies (adding **`check False`** statements, removing key specification elements to ascertain whether they are indeed used in a proof, and so on). Users with less experience would greatly benefit from tool support to detect possible inconsistencies and to track them back to specific annotations that ought to be rectified.[12]

Improving the documentation and learning material is another way of improving the usability of AutoProof for non-experts. We created AutoProof's tutorial with

[12] A simple way to implement support of this kind could build atop Boogie's smoke testing functionality.

an eye on the major difficulties the students encountered while working on their projects. Future iterations of the course will help us assess to what extent learning AutoProof has become easier for beginners.

7 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 preprocessor 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 [17], 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. 3.2, 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 [47]. As future work we plan to further experiment with integrating the feedback of diverse code analysis tools (AutoProof being one of them) to improve the usability of verification.

References

1. W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, number 8471 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, 2014.
2. 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/`.
3. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.
4. B. Beckert, D. Bruns, V. Klebanov, C. Scheben, P. H. Schmitt, and M. Ulbrich. Information flow in object-oriented software. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR*, volume 8901 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2014.
5. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
6. 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`.
7. 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/`.
8. J. M. Chimento, W. Ahrendt, G. J. Pace, and G. Schneider. StaRVOOrS: A tool for combined static and runtime verification of Java. In E. Bartocci and R. Majumdar, editors, *Runtime Verification – 6th International Conference, RV 2015*, volume 9333 of *Lecture Notes in Computer Science*, pages 297–305. Springer, 2015.
9. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
10. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: a practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. `http://vcc.codeplex.com/`.
11. D. Cok. The OpenJML toolset. In *NASA Formal Methods*, volume 6617. 2011.
12. Á. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499, 2008.
13. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
14. EiffelBase2: A fully verified container library. `https://github.com/nadia-polikarpova/eiffelbase2`, 2015.
15. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: overview and VerifyThis competition. *International Journal on Software Tools for Technology Transfer*, 17(6):677–694, 2015.

16. 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/.
17. 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/.
18. 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.
19. C. A. Furia. Rotation of sequences: Algorithms and proofs. http://arxiv.org/abs/1406.5453, June 2014.
20. C. A. Furia, C. M. Poskitt, and J. Tschannen. The AutoProof verifier: Usability by non-experts and on standard code. In C. Dubois, P. Masci, and D. Mery, editors, *Proceedings of the 2nd Workshop on Formal Integrated Development Environment (F-IDE)*, volume 187 of *Electronic Proceedings in Theoretical Computer Science*, pages 42–55. EPTCS, June 2015. Workshop co-located with FM 2015.
21. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
22. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis verification competition. http://verifythis2012.cost-ic0701.org, 2012.
23. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis verification competition. http://etaps2015.verifythis.org/, 2015.
24. 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/.
25. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VS-Tools workshop at VSTTE*, 2010.
26. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.
27. 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.
28. 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/.
29. 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.
30. 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.
31. K. R. M. Leino. This is boogie 2. Technical report, Microsoft Research, June 2008. http://research.microsoft.com/apps/pubs/default.aspx?id=147643.
32. 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/.
33. K. R. M. Leino and M. Moskal. Usable auto-active verification. In *Usable Verification Workshop*. http://fm.csl.sri.com/UV10/, November 2010.
34. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP 2004 – Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14–18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, 2004.
35. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.
36. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 246–257, 2002.
37. 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.
38. The OpenJML toolset. http://openjml.org/, 2013.
39. E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 46, 2014.
40. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *VSTTE*, volume 6217 of *LNCS*, pages 127–141. Springer, 2010.
41. N. Polikarpova, J. Tschannen, and C. A. Furia. A fully verified container library. In *FM*, LNCS. Springer, 2015.
42. 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.
43. SAVCBS workshop series. http://www.eecs.ucf.edu/~leavens/SAVCBS/, 2010.
44. A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *IWACO*, pages 1–9. ACM, 2009.
45. 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/.
46. J. Tschannen, C. A. Furia, and M. Nordio. AutoProof meets some verification challenges. *International Journal on Software Tools for Technology Transfer*, 17(6):745–755, 2015.
47. 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.
48. 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.

49. 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.
50. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In C. Baier and C. Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, April 2015.
51. 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.
52. S. West, S. Nanz, and B. Meyer. Efficient and reasonable object-oriented concurrency. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '15)*. ACM, 2015.