https://doi.org/10.1007/s00165-017-0435-1

Formal Aspects of Computing (2018) 30: 495–523

Formal Aspects of Computing

A fully verified container library

Nadia Polikarpova[1], Julian Tschannen[2], and Carlo A. Furia[3]

[1] MIT CSAIL, Cambridge, MA, USA
[2] Google, Zurich, Switzerland
[3] Chalmers University of Technology, Göteborg, Sweden

Abstract. The comprehensive functionality and nontrivial design of realistic general-purpose container libraries pose challenges to formal verification that go beyond those of individual benchmark problems mainly targeted by the state of the art. We present our experience verifying the full functional correctness of EiffelBase2: a container library offering all the features customary in modern language frameworks, such as external iterators, and hash tables with generic mutable keys and load balancing. Verification uses the automated deductive verifier AutoProof, which we extended as part of the present work. Our results indicate that verification of a realistic container library (135 public methods, 8400 LOC) is possible with moderate annotation overhead (1.4 lines of specification per LOC) and good performance (0.2 s per method on average).

Keywords: Deductive verification, SMT, Object-oriented software, Containers, AutoProof

1. Introduction

The moment of truth for software verification technology comes when it is applied to realistic programs in practically relevant domains. Libraries of general-purpose data structures—called *containers*—are a prime example of such domains, given their pervasive usage as fundamental software components. Data structures are also "natural candidates for full functional verification" [ZKR08] since they have well-understood semantics and typify challenges in automated reasoning such as dealing with aliasing and the heap. This paper presents our work on verifying full functional correctness [1] of a realistic, object-oriented container library.

1.1. Challenges

Realistic software has nontrivial size, a design that promotes flexibility and reuse, and an implementation that offers competitive performance. *General-purpose* software includes all the functionalities that users can reasonably expect, accessible through uniform and rich interfaces. *Full specifications* completely capture the behavior of a software component relative to the level of abstraction given by its interface. Notwithstanding the vast amount

Work mainly done while the authors were affiliated with ETH Zurich, Switzerland. A preliminary version appeared in the 20th International Symposium on Formal Methods in 2015 [PTF15].

Correspondence and offprint requests to: C.A. Furia, Email: furia@chalmers.se

[1] "Functional correctness" refers to the input/output behavior of programs [DB82, HLL+12].

of research on functional verification of heap-manipulating programs and its applications to data-structure implementations, no previous work has, to our knowledge, tackled all these challenges in combination.

Rather, the focus has previously been on verifying individually chosen data-structure operations, often stripped or tailored to particular reasoning techniques. Some concrete examples from recent work in this area (see Sect. 5 for more): Zee et al. [ZKR08] verify a significant selection of complex linked data structures but not a complete container library, and they do no include certain features expected of general-purpose implementations, such as iterators or user-defined key equivalence in hash tables. Pek et al. [PQM14] analyze realistic implementations of linked lists and trees but do not always verify full functional correctness (for example, they do not prove that reversal procedures actually reverse the elements in a list), nor can their technique handle arbitrary heap structures. Kawaguchi et al. [KRJ09] verify complex functional properties but their approach targets functional languages, where the abstraction gap between specification and implementation is narrow; hence, their specifications have a different flavor and their techniques are inapplicable to object-oriented designs. These observations do not detract from the value of these works; in fact, each challenge is formidable enough in its own right to require dedicated focused research, and all are necessary steps towards verifying realistic implementations—which has remained, however, an outstanding challenge.

1.2. Result

Going beyond the state of the art in this area, we completely verified a realistic container library, called EiffelBase2, against full functional specifications. The library, described in Sect. 4, consists of over 8000 lines of Eiffel code in 46 classes, and offers arrays, lists, stacks, queues, sets, and tables (dictionaries). EiffelBase2's interface specifications are written in first-order logic and characterize the abstract object state using mathematical entities, such as sets and sequences. To demonstrate the usefulness of these specifications for clients, we also verified correctness properties of around 2700 lines of client code that uses some of EiffelBase2's containers.

1.3. Techniques

A crucial feature of any verification technique is the amount of automation it provides. While some approaches, such as abstract interpretation, can offer complete "push button" automation by focusing on somewhat restricted properties, full functional verification of realistic software still largely relies on interactive theorem provers, which require massive amounts of effort from highly-trained experts. For example, verifying the seL4 OS kernel [KEH+09]—consisting of complex system-level code—took 20 person-years in Isabelle/HOL. Another example is Leroy's verification of a realistic compiler [Ler09]. Even data-structure verification uses interactive provers, such as in Zee et al. 's work [ZKR08], to discharge the most complex verification conditions. Advances in verification technology that target this class of tools have little chance of directly improving usability for *serious yet non-expert* users—as opposed to verification mavens.

In response to these concerns, an important line of research has developed verification tools that target expressive functional correctness properties, yet provide more automation and do not require interacting with back-end provers directly. Since their degree of automation is intermediate between fully automatic and interactive, such tools are called *auto-active* [LM10a]; examples are Dafny [Lei10], VCC [CDH+09], and VeriFast [JSP+11], as well as AutoProof, which we developed in previous work [PTFM14, TFNP15] and significantly extended as part of the work presented here.

At the core of AutoProof's verification methodology for heap-manipulating programs is *semantic collaboration* [PTFM14]: a flexible approach to reasoning about class invariants in the presence of complex inter-object dependencies. Previously, we applied the methodology only to a selection of stand-alone benchmarks; in the present work, to enable the verification of a realistic library, we extended it with support for custom mathematical types, abstract interface specifications, and inheritance. We also redesigned AutoProof's encoding of verification conditions in order to achieve *predictable* performance on larger problems. These improvements, which we describe in Sect. 3, directly benefit serious users of the tool by providing more automation, better user experience, and all-out support of object-oriented features as used in practice.

1.4. Contributions

This paper's work makes the following contributions:

- The first verification of full functional correctness of a *realistic general-purpose data-structure library* in a heap-based object-oriented language.
- The first verification of a significant collection of data structures carried out entirely using an *auto-active verifier*.
- The first full-fledged verification of several *advanced object-oriented patterns* that involve complex inter-object dependencies but are widely used in realistic implementations (see Sect. 2).
- A practical verification methodology and the supporting AutoProof verifier, which are suitable to reason, with *moderate annotation overhead* and *predictable performance*, about the full gamut of object-oriented language constructs.

The fully annotated source code of the EiffelBase2 container library and a web interface for the AutoProof verifier are available at:

https://github.com/nadia-polikarpova/eiffelbase2 (cite as [Pol15])

The paper focuses on presenting EiffelBase2's verification effort and the new features of AutoProof that we introduced to this end; to make the paper self-contained, we also include an overview of some components of AutoProof's verification methodology that had been also reported in our previous work but were crucial to EiffelBase2's verification, such as semantic collaboration [PTFM14] and model-based interface specifications [PFM10]. For an in-depth presentation of these techniques and of the rest of AutoProof's features, we refer readers to the related publications [PTFM14, PFM10, TFNP15, FNPT16, Pol14].

2. Illustrative examples

Using excerpts from two data structures in EiffelBase2—a linked list and a hash table—we demonstrate our approach to specifying and verifying full functional correctness of containers, and illustrate some challenges specific to realistic container libraries.

2.1. Linked list

Figure 1 shows excerpts from two EiffelBase2 classes that implement singly linked lists and their iterators. Class `LINKED_LIST` is parametrized by the generic type `G` of list elements, and inherits from an abstract class `LIST`, which defines the interface common to all three implementations of lists available in EiffelBase2—arrayed, singly linked, and doubly linked. As shown in Figure 5, class `LIST` in turn inherits indirectly from class `CONTAINER`, which is the root of EiffelBase2's container hierarchy. The excerpt in Fig. 1 includes two public API methods: `first`, for accessing the first element of a list, and `extend_back`, for inserting an element at the list's end. The excerpt also shows the private attributes of `LINKED_LIST` that make up its internal representation: `first_cell` stores a reference to the first node (for accessing the list's content), and `last_cell` stores a reference to the last node in the list (for making insertion at the end a constant-time operation). List nodes use a helper class `LINKABLE [G]` with two attributes: `item: G` (the value stored in the node) and `right: LINKABLE [G]` (a reference to the following node). The rest of `LINKED_LIST`'s attributes are there solely for specification purposes, hence their designation as `ghost` [FGP14].[2]

Class `LINKED_LIST_ITERATOR` implements iterator objects; any such object is attached to a given linked list upon construction (constructor method `make`), and can traverse the list (method `forth`) while accessing list elements (method `item`) as well as inserting or removing elements at the current iterator position (methods such as `remove_right`). An iterator's state consists of a publicly visible reference to the `target` list and a private reference `active` to the list node at the current position.

[2] Section 3.2 better explains the notion of ghost elements.

```
class LINKED_LIST [G] inherit LIST [G]
model sequence

feature {public}
  ghost sequence: MML_SEQUENCE [G]
  ghost bag: MML_BAG [G] -- inherited from CONTAINER

  first: G -- First element.
    require not sequence.is_empty
    do
      assert inv
      Result := first_cell.item
    ensure Result = sequence.first

  extend_back (v: G) -- Insert 'v' at the back.
    require all o ∈ observers : not o.closed
    modify model Current [sequence]
    local cell: LINKABLE [G]
    do
      create cell.put (v)
      if first_cell = Void then
        first_cell := cell
      else
        last_cell.put_right (cell)
      end
      last_cell := cell
      cells := cells + ⟨cell⟩
      sequence := sequence + ⟨v⟩
    ensure sequence = old sequence + ⟨v⟩

feature {private}
  first_cell: LINKABLE [G]
  last_cell: LINKABLE [G]
  ghost cells: MML_SEQUENCE [LINKABLE [G]]

invariant {public}
  seq_refines_bag: bag = sequence.to_bag
invariant {private}
  cells_domain: sequence.count = cells.count
  first_cell_empty: cells.is_empty = (first_cell = Void)
  last_cell_empty: cells.is_empty = (last_cell = Void)
  owns_definition: owns = cells.range
  cells_exist: cells.non_void
  sequence_implementation: all i ∈ 1 .. cells.count :
    sequence [i] = cells [i].item
  cells_linked: all i, j ∈ 1 .. cells.count :
    i + 1 = j implies cells [i].right = cells [j]
  cells_first: cells.count > 0 implies
    first_cell = cells.first
  cells_last: cells.count > 0 implies
    last_cell = cells.last and last_cell.right = Void
end
```

```
class LINKED_LIST_ITERATOR [G] inherit LIST_ITERATOR [G]
model target, index

feature {public}
  target: LINKED_LIST [G]
  ghost index: INTEGER

  make (list: LINKED_LIST [G]) -- Constructor.
    modify Current
    modify field list [observers, closed]
    do
      target := list
      target.add_iterator (Current)
      assert target.inv_only (seq_refines_bag)
    ensure
      target = list
      index = 0
      list.observers = old list.observers + {Current}

  item: G -- Item at current position.
    require not off and all s ∈ subjects : s.closed
    do
      assert inv and target.inv
      Result := active.item
    ensure Result = target.sequence [index]

  forth -- Move one position forward.
    require not off and all s ∈ subjects : s.closed
    modify model Current [index]
    do ...
    ensure index = old index + 1

  remove_right -- Remove element after the current.
    require
      1 ≤ index ≤ target.sequence.count − 1
      target.is_wrapped -- closed and owner = Void
      all o ∈ target.observers :
        o ≠ Current implies not o.closed
    modify model target [sequence]
    do ...
    ensure target.sequence =
      old target.sequence.removed_at (index + 1)

feature {private}
  active: LINKABLE [G]

invariant {public}
  target_exists: target ≠ Void
  subjects_definition: subjects = {target}
  index_range: 0 ≤ index ≤ target.sequence.count + 1
invariant {private}
  cell_off: (index <1 or target.sequence.count <index)
    = (active = Void)
  cell_not_off: 1 ≤ index ≤ target.sequence.count
    implies active = target.cells [index]
end
```

Fig. 1. Excerpt from EiffelBase2 classes `LINKED_LIST` and `LINKED_LIST_ITERATOR`. (The syntax is occasionally simplified for readability.)

The Eiffel language includes notation for method preconditions (`require`) and postconditions (`ensure`), and class `invariant`s; AutoProof provides additional keywords to annotate methods with frame conditions (`modify` and `read`), which express what a method's body may modify or access. Every specification element (such as a precondition or a class invariant) consists of one or more clauses (assertions), which are implicitly conjoined. A clause may have a name (tag), which carries no semantics but is useful to refer to specific parts of a large specification element, in particular in error messages. For example, `LINKED_LIST`'s class invariant has ten clauses; the first one is named `seq_refines_bag`.

2.1.1. Interface specifications

An important goal when specifying libraries is simplifying client reasoning: the interface specification must, on the one hand, hide from clients all the *irrelevant* details, such as the intricate heap structure underlying a container's implementation; on the other hand, it must provide enough *relevant* details for clients to reason, formally and informally, about whether they are interacting with the container correctly and what the observable effect of their interactions is. In EiffelBase2 we meet these diverging requirements with a specification methodology that explicitly distinguishes between *concrete* and *abstract* state of a class; the specification of the publicly observable behavior must be as precise as possible, but only relative to the information captured by the abstract state.

Each EiffelBase2 class declares its abstract state as a set of `model` attributes. In the case of `LINKED_LIST`, it is a single attribute `sequence`, which denotes the sequence of list elements; its type `MML_SEQUENCE` is not a normal Eiffel class, but a so-called *model class* (which defines a *specification type*) from the Mathematical Model Library (MML). Instances of model classes are mathematical values with custom logical representations in the underlying prover. Since attribute `sequence` is declared as `ghost`, it exists only for specification and verification purposes but is stripped away by the compiler; thus, any elements introduced for specification purposes bring no runtime overhead.

It is good practice—uniformly accepted in Eiffel [Mey97]—to separate methods into *commands* and *queries*; EiffelBase2 method specifications reflect this command/query separation. Commands—methods with observable side effects that return no value—modify the abstract state of objects listed in their frame specification according to their postcondition. For example, the `ensure` clause of command `extend_back` in Fig. 1 states that its effect on the abstract state is appending its argument to the lists's sequence of elements; at the same time, its `modify` clause implies that this will be the only observable side effect: model attribute `sequence` is the only piece of visible abstract state that `extend_back` may modify. Queries—methods that return a value without observable side effects—specify, in their postcondition, their returned value as a function of the abstract state, which they do not modify. For example, the `ensure` clause of query `first` in Fig. 1 states that it simply returns the first element in the list's sequence of elements.

By referring to an explicitly declared model, interface specifications are concise, have a consistent level of abstraction, and can be checked for *completeness*, that is whether they uniquely characterize the results of queries and the effect of commands on the model state. Such abstract specifications are convenient for clients, which can reason about the effect of method calls solely in terms of the model while ignoring implementation details. Indeed, `LINKED_LIST`'s public specification is the same as `LIST`'s—its abstract ancestor class—and is oblivious to the fact that the sequence of elements is stored in linked nodes on the heap.

While clients have it easy, verifying different implementations of the same abstract interface poses additional challenges in ensuring consistency without compromising on individual implementation features.

2.1.2. Connecting abstract and concrete state

Verifying the implementation of `first` in Fig. 1 requires relating the model of the list to its concrete representation: more precisely, specifying that the first element of the abstract `sequence` is the same as the `item` stored in the first node of the list, referenced by `first_cell`. We accomplish this through the class `invariant`: the clause named `sequence_implementation` asserts that model attribute `sequence` lists the items stored in the sequence of `LINKABLE` nodes denoted as `cells`; `cells` is an auxiliary specification-only attribute connected to the concrete list representation by invariant clauses `cells_first` and `cells_linked`, which together say that one can traverse `cells` (when it's not empty) by following `right` links starting at `first_cell`.

2.1.3. Invariant methodology

Reasoning based on class invariants is germane to object-oriented programming, yet the semantics of invariants is tricky. A fundamental issue is *when* (at what program points) invariants should hold. Simple syntactic approaches, which require invariants to hold at predefined points (for example, before and after every public call), are not flexible enough to reason about complex object structures. Our methodology is based on the notion of *closed* (and open) objects [LM04, BN04]: invariants have to hold only for closed objects. Programs modify objects following a protocol that requires opening an object, changing its state, and then closing it, while providing evidence that the invariant has been restored; the only way to open and close an object is through built-in methods `unwrap` and `wrap`

respectively. By default, AutoProof implicitly opens the `Current` object[3] at the beginning of every public command and closes it at the end: hence the absence of explicit annotations in the body of `extend_back`; this behavior can be easily overridden when more flexibility is necessary.

2.1.4. Ownership

`LINKED_LIST`'s invariant relies on the content of its `cells`. This might threaten modularity of reasoning, since an independent modification of a cell by an unknown client may break consistency of the list object. The developer's intent, however, is that the cells be part of the list's internal representation, which should not be directly accessible to other clients; a verifier must provide means to both state and formally verify this intent. For such *hierarchical* object dependencies, AutoProof implements an *ownership* scheme [LM04, CDH+09] to specify which objects have unique control over which other objects. `LINKED_LIST`'s invariant clause `owns_definition` asserts that the list owns precisely its cells; the owner is the gatekeeper to any modification of the `cells`' content, which is then only possible when the owner is open.

2.1.5. Safe iterators

Like standard container libraries in other languages, EiffelBase2 offers iterator classes, which provide the most idiomatic and uniform way of manipulating containers (in particular, lists). When multiple iterators are active on the same list, consistency problems may arise: modifying the list, through its own interface or one of the iterators, may invalidate the other iterators. Concretely, consider the invariant clause `cell_not_off` of `LINKED_LIST_ITERATOR` in Fig. 1: it connects the abstract state of the iterator—the integer `index` of its current position inside the list—with its concrete state—the reference `active` to the current list node. If two iterators are positioned on consecutive elements of the same list, calling `remove_right` on the leftmost iterator removes the `active` node of the other iterator, and thus indirectly violates the latter's invariant clause `cell_not_off`.

Such scenarios are not only a challenge to verification but practical programming problems: even in the absence of formalized invariants, a client would observe undesired behavior (namely, an iterator's `item` method returning a value that is no longer in the list). To address it, Java's `java.util` iterators implement *fail-safe* behavior, which amounts to equipping containers with a `version` attribute, checking its value at every iterator usage, and raising an exception if that value has changed. This is not a robust solution, since "the fail-fast behavior of an iterator cannot be guaranteed", and hence one cannot "write a program that [depends] on this exception for its correctness" [Jav16a]. Other collection libraries, such as .NET's Generic Collections, choose to provide no protection and rely on the programmer's informal reasoning, who should be aware that "if changes are made to the collection, such as adding, modifying, or deleting elements, the [iterator's] behavior is undefined" [.NE16b]; to reduce the resulting ample possibility for human error, the .NET library restricts the interface of its iterators, preventing them from modifying the underlying collections.

In contrast, through complete specifications, EiffelBase2 offers robust *safe* iterators: clients reason about correct usage statically, so that safe behavior can be guaranteed without runtime overhead or functionality restrictions. To enable this, once again, a verifier must allow developers to precisely formalize and verify their *intent*: namely, that a container is allowed to have either multiple active reader-iterators or a single active writer-iterator (or no iterators, in which case clients are free to modify the container through it own interface).

2.1.6. Collaborative invariants

Object dependencies such as those arising between a list and its iterators do not quite fit hierarchical ownership schemes (Sect. 2.1.4): an iterator's consistency depends on the list, but any one iterator cannot own the list—simply because other iterators may be active on the same list. In such cases we rely on *collaborative invariants*, introduced in our previous work [PTFM14], to specify which objects depend on which other objects for consistency. `LINKED_LIST_ITERATOR`'s invariant clause `subjects_definition` asserts that the iterator might depend on its `target` list (one of its `subjects`); correspondingly, the list has to include all active iterators among its `observers` when they register upon construction by calling `add_iterator`. By reasoning about the objects in sets `subjects` and `observers`, we ascertain that closed iterators remain valid.

[3] Keyword `Current` is the Eiffel equivalent of `this` in Java and similar languages; it is a reference to the *receiver* object within the object's class text.

```
class HASH_TABLE [K, V]
model map, lock

feature {public}
  ghost map: MML_MAP [K, V]
  ghost lock: LOCK [K]

  extend (k: K; v: V) -- Add key-value pair.
    require
      k ∈ lock.owns
      all x ∈ map.domain : not x.is_equal (k)
      lock.is_wrapped
      observers = {}
    modify model Current [map]
    do ...
    ensure map = old map.updated (k, v)

feature {private}
  buckets: ARRAY [LINKED_LIST [PAIR [K, V]]]

invariant {public}
  subjects_definition: subjects = {lock}
  keys_locked: map.domain ≤ lock.owns
  no_duplicates: all x, y ∈ map.domain :
    x ≠ y implies not lock.eq [x, y]
invariant {private}
  keys_in_buckets: all x ∈ map.domain :
    buckets [index (lock.hash [x])].has (x)
end
```

```
ghost class LOCK [K]
model eq, hash

feature {public}
  eq: MML_RELATION [K, K]
  hash: MML_MAP [K, INTEGER]

  lock (key: K) -- Acquire ownership of 'key'.
    require key.is_wrapped
    modify Current
    modify field key [owner]
    do ...
    ensure owns = old owns + {key}

  unlock (key: K) -- Relinquish ownership of 'key'.
    require
      key ∈ owns
      all o ∈ observers : not key ∈ o.map.domain
    modify Current
    do ...
    ensure
      owns = old owns − {key}
      key.is_wrapped

invariant {public}
  eq_definition: all x, y ∈ owns : eq [x, y] = x.is_equal (y)
  hash_definition: all x ∈ owns : hash [x] = x.hash_code
end
```

Fig. 2. Excerpts from classes HASH_TABLE and LOCK. (The syntax and specifications are occasionally simplified for readability.)

For example, the precondition of method `remove_right` in class LINKED_LIST_ITERATOR requires that all other observers of the `target` list apart from the receiver be open, which enables updating the list's state without running the risk of breaking invariants of closed objects. At the same time, methods like `item` and `forth` do not impose this requirement, which allows multiple closed iterators to read from the same list. Thus collaborative invariants readily support formalization of the intended iterator behavior outlined in the previous paragraph.

2.2. Hash table

Figure 2 shows excerpts from EiffelBase2's class HASH_TABLE, alongside the auxiliary specification-only class LOCK. Method `extend` inserts a new key/value pair into a hash table; its effect is specified in terms of model attribute `map`, which abstracts the state of the hash table as a finite mathematical map from keys to values. The private representation of a hash table is an array of `buckets`; following a traditional open hashing (chaining) implementation [CLRS09, Sec. 12.2], each bucket is a linked list of key/value pairs. The key property of a hash table, which makes lookup efficient, is formalized by invariant clause `keys_in_bucket`: each key in `map`'s domain is stored in the bucket designated by the hash function `hash`.

2.2.1. Custom mutable keys

As in any realistic container library, EiffelBase2's hash tables support arbitrary objects as keys, with user-defined equivalence relations and hash functions. For example, a class BOOK might override the `is_equal` method (`equals` in Java) to compare two books by their ISBN, and define `hash_code` accordingly. When a table compares keys by object content rather than by reference, changing the *state* of an object used as key may break the table's consistency. Libraries without full formal specifications cannot precisely characterize such unsafe key modifications; Java's `java.util` maps, for example, generically recommend "great care [if] mutable objects are used as map keys", since "the behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map" [Jav16b]; similarly, .NET's Generic Collections' dictionaries require that "As long as an object is used as a key in the dictionary, it must not change in any way that affects its hash value" [.NE16a]. In contrast, EiffelBase2's specification precisely captures which key modifications affect consistency, ensuring safe behavior without restricting usage scenarios.

```
test
  local
    lock: LOCK
    location: HASH_TABLE [BOOK, STRING]
    copies: HASH_TABLE [BOOK, INTEGER]
    b: BOOK
  do
    -- Create two tables with a shared lock:
    create lock
    create location.make (lock)
    create copies.make (lock)
    ...
    -- Add the same book to both tables:
    create b.with_isbn (1840226358)
    lock.lock (b)
    location.extend (b, "Aisle 3")
    copies.extend (b, 10)
    ...
    -- Modify the book used as key:
    lock.unwrap
    b.set_year (1922)
    lock.wrap
  end
```

Fig. 3. Example usage of HASH_TABLE in client code

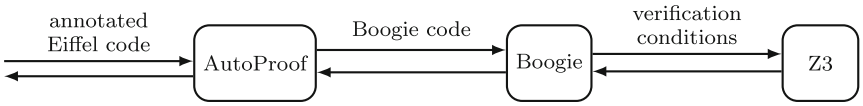

Fig. 4. AutoProof's toolchain

2.2.2. *Shared ownership*

A table's consistency depends on its keys, but this dependency fits neither ownership nor collaboration straightforwardly: keys may be shared between tables, and hence any one table cannot own its keys; collaboration would require key objects to register their host tables as `observers`, thus preventing the use of independently developed classes as keys. In EiffelBase2, we address these challenges by means of a *shared ownership* specification pattern that combines ownership and collaboration. A class LOCK (outlined in Fig. 2) acts as an intermediary between tables and keys: it owns keys and maintains a summary of their relevant properties (their hash codes and the equivalence relation induced by `is_equal`); multiple tables observe a single LOCK object and rely on its summary instead of directly observing keys. For example, the code in Fig. 3 demonstrates how an EiffelBase2 client, which is part of a book store information system, can use a single BOOK object as a key in two different hash tables, `location` and `copies`.

Figure 3 also shows how clients can modify keys that are in use by one or more hash tables, as long as the invariant of the keys' lock is maintained. In order to invoke `set_year` on key `b`, the client must open its owner, `lock`; closing back the lock after performing the modification requires proving its invariant, which is only possible if `set_year` is guaranteed to leave `b`'s ISBN unchanged. Therefore, if the client code in Fig. 3 verifies, we can be sure that the new state of `b` is still consistent with the `lock`'s summary of its equivalence class and hash code, and thus no hash tables have been made inconsistent by the change.

LOCK is a `ghost` class: all its state and operations are absent from the compiled code. The actual implementation of LOCK in EiffelBase2 is more general than the simplified version presented in Fig. 2; in particular, it is usable with structures other than hash tables, and uses an advanced feature of semantic collaboration called *update guards* [PTFM14] to support locking and unlocking objects without notifying the lock's observers, which helps limit manual annotation overhead.

3. Verification approach

AutoProof works by translating annotated Eiffel code into the Boogie intermediate verification language [BCD+05], and uses the Boogie verifier to generate verification conditions, which are then discharged by the SMT solver Z3 [dMB08] (see Fig. 4). As part of verifying EiffelBase2 we extended the *verification methodology* of AutoProof (the tool's underlying logic) and substantially redesigned its *Boogie encoding*. This section describes the most significant aspects of the improved methodology and tool, which the examples of Sect. 2 informally introduced; we refer to both the methodology and the tool simply as "AutoProof".

3.1. Specification types

AutoProof offers a Mathematical Model Library (MML) of specification types: sets, bags (multisets), pairs, relations, maps, and sequences. Each type corresponds to a *model class* [Cha06]: a purely applicative class whose semantics for verification is given by a collection of type signatures, function signatures, and axioms in Boogie. Unlike verifiers that use built-in syntax for specification types, AutoProof is *extensible* with new types by providing an Eiffel wrapper class and a matching Boogie theory, which can be used like any existing MML type. The MML implementation used in EiffelBase2 relies on 228 Boogie axioms (see [Pol14, Ch. 6] for details); most of them have been borrowed from Dafny's background theory, whose broad usage supports confidence in their consistency.

3.2. Ghost state

Auto-active verification commonly relies on *ghost state*—variables that are only mentioned in specifications and do not affect executable code—as its main mechanism for abstraction. This is different from specialized mechanisms such as *model variables* [CLSE05, LBR06, Lei95, Mül02] used, for example, in JML. A model variable is a special kind of attribute that cannot be directly modified; its value gets updated automatically with every modification of the concrete state so as to satisfy a special invariant called the *abstraction function*.[4] The model variable approach offers the advantage of not requiring explicit updates to keep abstract and concrete state in sync; however, it also has known soundness and modularity issues [LM06]. First, it often happens that the abstraction is well-defined if the concrete state is consistent, but may become undefined during intermediate states in an update—when the class invariant is temporarily violated; in these cases, the constraints that regulate the automatic update of model variables may become unsatisfiable, thus introducing unsoundness in the form of assuming an unsatisfiable predicate. Second, updates to the concrete state may have *instant effects* on the values of model variables elsewhere in the system, which is detrimental to modularity.

To preserve simplicity while avoiding these issues, AutoProof follows other practical auto-active tools, such as Dafny and VCC, and encodes abstractions as regular [5] variables and attributes, with the relation between abstract and concrete states given in the class invariant. Since AutoProof only requires invariants to hold for closed objects (see Sect. 3.4), this treatment naturally supports partial abstraction functions, while also giving methods the freedom to modify an object's abstract and concrete states independently when the object is open. Under this scheme, unsatisfiable constraints on the abstract state are easily detected, since they prevent an object from ever becoming closed.

The downside of AutoProof's approach is that it requires explicit updates to the ghost state inside method bodies. Explicit update code quickly proliferates in realistic programs with many ghost variables such as EiffelBase2, even if it generally remains straightforward because the relation between abstract and concrete program state tends to be simple. For an example look at `LINKED_LIST`'s ghost attribute `bag` in Fig. 1: the attribute is inherited from class `CONTAINER`, where it is a model attribute (a generic container does not impose order on elements); in `LIST`, however, `bag` becomes subsumed by the new model attribute `sequence`, and in fact the invariant clause `seq_refines_bag` defines `bag` in terms of `sequence`. Such a tedious pattern of "legacy" ghost attributes requiring explicit updates is recurrent in EiffelBase2 because of its rich inheritance hierarchy.

To assuage this common problem, AutoProof offers *implicit updates* for ghost attributes: for every class invariant clause `ga = expr` that relates a ghost attribute `ga` to an expression `expr`, AutoProof implicitly adds the assignment `ga := expr` just before the receiver object transitions from open to closed state. In Fig. 1, for example, invariant clause `seq_refines_bag` gives rise to the implicit assignment `bag := sequence.to_bag` at the end of `extend_back`; this has the effect of automatically keeping the inherited attribute (`bag`) in sync with its refinement (`sequence`). We believe that implicit attributes successfully address the excess annotation overhead of ghost state in simple cases, while retaining the conceptual simplicity and flexibility that are required in more sophisticated scenarios.

[4] As a side remark, the term "abstraction function" is a bit of a misnomer, since model variables are deliberately meant to support *relational* rather than (more common) functional specifications.

[5] The distinction between ghost and concrete state only exists for the compiler, while the verifier treats both in the same way.

3.3. Model-based specifications

As illustrated in Sect. 2, each class specification includes a `model` clause, which designates a subset of attributes of the class as the class *model*. The model precisely defines the publicly observable *abstract state* of the class, on which clients solely rely. Method postconditions refer to the model: *commands* (such as `extend_back` in Fig. 1) relate the model post-state to the model pre-state (referenced by `old` expressions); *queries* (such as `first` in Fig. 1) relate the returned `Result` to the model state.

Model attributes and ghost attributes (Sect. 3.2) are orthogonal concepts; in particular, models may include concrete (non-ghost) attributes whenever implementation elements are also part of the interface (as is the case for `target` in Fig. 1).

While abstracting object state is a common technique in formal methods, the model-based specification discipline we advocate here has additional advantages: it upholds rigorous and actionable quality criteria, such as *completeness* [PFM10] and *observability* [WEH+96]. Informally, 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 uniquely defines the returned result in terms of the model; the model of a class `C` is complete if it supports complete specifications of all public methods in `C`. For example, a set is not a complete model for `LINKED_LIST` in Fig. 1 because the precise result of `first` cannot be defined as a function of a set.

If a method's specification is complete (with respect to a chosen mathematical model), clients can reason about any effects the method has on the state abstracted by the model. An incomplete method specification calls for strengthening the method's postcondition; when this turns out to be impossible, we have a a strong indication that the model was chosen poorly. While complete specifications should be the norm, there are situations where incomplete ones are preferable or even necessary; these include inherently *nondeterministic* behavior—such as methods reading input from the outside world—and *partial abstractions*—resulting from using inheritance to factor out common parts of individually complete specifications. Even when incompleteness turns out to be necessary, the very act of assessing the completeness of specifications is still likely to improve the understanding of the programs being written and to promote rigorous, motivated design choices regarding interfaces and inheritance hierarchies.

The model of a class is *observable* if any two different model states are distinguishable by the results returned by calling public methods of the class. For example, a sequence is not an observable model of a class `CONTAINER` (mentioned in Sect. 2.1) because a container's interface provides no methods that discriminate element ordering, and hence two sequences with the same elements in different order are indistinguishable. Weide et al. [WEH+96] discuss the importance of observability for understandable specifications; most important, non-observable specifications are confusing for clients, because they introduce immaterial distinctions in the model. When a class specification is complete but not observable, we may want to reconsider our choice of model or add more features to the class interface.

In related work [Pol14, Ch. 3] we present a formalization of completeness and observability within the theory of abstract data types, and introduce an encoding of these properties for checking them mechanically. AutoProof currently does not implement this encoding; however, we find that even reasoning informally about completeness and observability—as we did sweepingly in the design of EiffelBase2—helps provide clear guidelines for writing interface specifications and substantiates the notion of "full functional correctness".

3.4. Class invariants

Class invariants are central to AutoProof's verification approach, since they can naturally express idiomatic object-oriented patterns and support succinct client reasoning. In EiffelBase2, they provide three kinds of specification: *abstract* invariants are interface specifications that constrain the model state of valid objects (for example, clause `index_range` in Fig. 1); *representation* invariants connect abstract and concrete state (for example, `cells_first` in Fig. 1); *linking* invariants define legacy model attributes, which are inherited but no longer directly used, in terms of the current model (for example, `bag_definition` in Fig. 1, where `bag` is the model of ancestor class `CONTAINER`).

Visibility Out of the three categories of invariant clauses, abstract and linking invariants are part of the public interface of the class, while representation invariants are hidden from clients. In Figure 1 and 2, in the interest of readability, we introduced visibility annotations on `invariant` blocks in order to distinguish between public and private invariant clauses. The actual Eiffel syntax does not offer such an annotation kind; instead, the visibility of a specification clause (including invariants) is determined by the visibility of the methods and attributes it mentions; users can rely on editors or IDEs to generate different views of the class code.

However, Eiffel has a more fine-grained visibility mechanism than most object-oriented languages: in addition to being completely public or completely private, an attribute (and hence a specification clause) may be *selectively exported*, that is visible only to a user-defined set of classes. This mechanism is well aligned with AutoProof's notion of collaboration: the representation-specific attributes and invariants of the class `LINKED_LIST`, for simplicity denoted as `private` in Fig. 1, are in fact selectively exported to `LINKED_LIST_ITERATOR`, which can then refer to them in its invariants and implementation.

Inconsistent states To precisely and flexibly define when invariants should hold, AutoProof follows the Spec# approach [LM04, BN04] of equipping every object with a built-in ghost Boolean attribute `closed`. Whenever an object is closed (`closed` is true), its invariant has to hold; but when it is open (`closed` is false) the invariant is allowed to be violated. The only way to update `closed` is through built-in ghost methods `unwrap` and `wrap`: `unwrap` opens a closed object, which becomes available for modification; `wrap` closes an open object provided its invariant holds. To reduce manual annotations, AutoProof adds calls to `Current.unwrap` and `Current.wrap` respectively at the beginning and end of every public command, and adds `Current.closed` to its pre- and postcondition; defaults can be overridden to implement different behavior.

This type of invariant reasoning fosters a flexible decoupling between knowing that an object is consistent and knowing details of its invariant. This is a powerful abstraction mechanism: it simplifies client reasoning, enables reusing inherited implementations and proofs (Sect. 3.6), and supports a more efficient Boogie encoding (Sect. 3.7). A similar decoupling is available in separation logic through abstract predicates and abstract predicate families [PB08], while, to our knowledge, it hasn't been incorporated into verification methodologies based on dynamic frames [Kas06] [6].

Dependent invariants Class invariants may involve multiple objects that are part of a coherent object structure (such as a linked list and its nodes, or an iterator and its target container). Such dependencies challenge invariant reasoning: how can one guarantee that the invariants of all closed objects hold, given that updating an object y could break the invariant of a dependent closed object x that y might not even know about? To address this issue AutoProof implements an invariant methodology called *semantic collaboration*, which supports sound and flexible reasoning about dependent invariants.

3.4.1. Hierarchical object structures

For hierarchical object dependencies, AutoProof provides an *ownership* model, closely following VCC's [CDH+09]. Each object x includes a built-in ghost attribute `owns`, which stores the set of "owned" objects y on which x may depend. AutoProof enforces that y cannot be opened (and thus modified) while x is closed, thus preventing indirectly breaking the consistency of x.

EiffelBase2's `LINKED_LIST`—a clear example of a hierarchical object dependency—relies on a chain of linked nodes as part of its internal representation and refers to their state (both their stored `item`s and their `right` links) in its invariant clauses `sequence_implementation`, `cells_linked`, and `cells_last`. AutoProof deems these clauses *admissible* thanks to another clause, `owns_definition`, which stipulates that all elements of the `cell` sequence are members of the list's `owns` set. Modifying the state of a node, such as inside the call to `last_cell.put_right` in `extend_back`, requires that the node be open; at the same time, unwrapping any object requires that its owner be open; thus it is not possible to manipulate the state of the list's nodes—and violate its invariant—while the list is closed.

[6] Note that simply wrapping an invariant definition in a predicate does not quite achieve decoupling; in order to make full use of such a decomposition, the verifier must be able to distinguish between modules that need to know the predicate definition for their verification and those that do not; for the latter, proofs can be reused even when the predicate definition changes (for instance, due to inheritance).

The definition of owns is fairly straightforward in LINKED_LIST, but supporting a wide variety of hierarchical object structures requires in general the full expressive power of first-order logic. A *semantic* approach like AutoProof's provides such unrestricted expressiveness to specify the structure of owns even in cases that were precluded to previous, more rigid, syntactic approaches [LM04].

3.4.2. Collaborative object structures

In realistic object-oriented programs not all dependencies are hierarchical: sometimes objects on the same level of abstraction collaborate to achieve a shared goal. For such "flat" object dependencies AutoProof provides a *collaboration* model, proposed in our previous work [PTFM14]. Each object x includes built-in ghost attributes subjects and observers; subjects is the set of objects x may depend on, and observers is the set of objects that may depend on x. AutoProof checks that subjects and observers are consistent across dependent objects (that is, any subject y of x contains x in its observers set), and that every update to a subject does not affect the consistency of its observers.

The dependency between a linked list and its iterators provides a nice illustration of collaborative object structures. Similarly as in ownership, the content of sets subjects and observers may be defined in the invariants. The invariant of LINKED_LIST_ITERATOR specifies that the target list is one of the iterator's subjects (clause subjects_definition); the following invariant clauses can depend on the state of the target thanks to this declaration of subjects. Whenever the list changes its state, such as when executing extend_back, it has to guarantee that all its observers are open and hence won't be invalidated by the state change; indeed, the precondition of extend_back fulfills this guarantee.

In addition, soundness requires that the observers set of a list object l contain all closed iterator objects that include l among their subjects. A key insight of semantic collaboration is that such consistency between subjects and observers can be guaranteed simply by implicitly adding a clause all s ∈ subjects : Current ∈ s.observers to the invariant of every class. Upon construction of an iterator object i, the clause forces the constructor LINKED_LIST_ITERATOR.make to add i to l's observers set by calling target.add_iterator. Afterwards, the same clause prevents l from removing i from its observers set: modifying l.observers, just like any other attribute, requires unwrapping i [7]; once unwrapped and removed from l.observers, i cannot be wrapped anymore, since the implicit clause of its invariant remains violated. In related work [PTFM14], we show that this treatment of collaborative invariants is extremely flexible and supports a wide variety of common object-oriented patterns; this is also confirmed by the present work, where semantic collaboration supported reasoning about object structures, such as iterators and hash tables with mutable keys, in a realistic setting.

3.5. Framing

Frame specifications—expressing which parts of the heap are modified by which methods—work hand in hand with the invariant methodology to provide effective local reasoning. A method m's frame specification modify $s_1, \ldots, s_n$—where, for $k = 1, \ldots, n$, each s_k denotes a set of objects—specifies that m is only allowed to modify objects y such that one of three conditions holds: (1) y is newly allocated by m's body, (2) $y \in s_1 \cup \cdots \cup s_n$, or (3) there exists an object $z \in s_1 \cup \cdots \cup s_n$ such that z transitively owns y. Automatically including transitively owned objects into method frames promotes information hiding: it is natural that modifying an object entails modifying other objects that are part of its internal representation, so this should be allowed by default; at the same time, clients should not rely on the state of such owned objects, nor have to check whether their state has changed. For example, Current is part of extend_back's frame specification in Fig. 1; thus, extend_back can update last_cell (one of the list's cells) because Current owns it.

For finer-grained frame specifications, AutoProof provides the syntax modify field s[$a_1, \ldots, a_n$]: a method with such a frame specification may only modify attributes $a_1, \ldots, a_n$ of the objects in set s. Model attributes play a special role in frame specifications: a method annotated with modify model s[$m_1, \ldots, m_n$] may only modify attributes $m_1, \ldots, m_n$ in the abstract state of the objects in set s, but has no direct restrictions on modifying the concrete state of any object in s (for example, method forth of LINKED_LIST_ITERATOR in Fig. 1 can modify Current.active but not Current.target). This construct enables fine-grained, yet abstract, frame specifications, in a way similar to data groups [LPZ02].

[7] Unless the attribute under modification has a nontrivial *update guard* [PTFM14]; in fact, in our implementation the update guard of the observers attribute makes it possible to only unwrap the observer objects that are being removed from the set.

3.6. Inheritance

Inheritance poses a challenge to modular verification methodologies, which have to find a good trade-off between permitting the descendant classes to extend and modify inherited specifications and implementations on the one hand, and maximizing code and proof reuse on the other hand. In this section we describe the solutions we chose for AutoProof, which we found particularly suitable for a reusable library with an extensive inheritance hierarchy such as EiffelBase2.

3.6.1. Invariants and inheritance

The interaction between invariants and inheritance has been the subject of much existing research [BDF+04, MPHL06, LW08, PB08]. The main challenge is that invariants are allowed to be strengthened in descendant classes, and as a result any method that was proven to (re-)establish the receiver's invariant, might not be doing the job any longer in the descendant. A simple solution—re-verifying the bodies of all inherited methods—is infeasible whenever the source code of the ancestor class is unavailable, and inefficient in any case.

Existing approaches to this problem [LM09, PB08] are based on the observation that inherited methods will not break the new invariant if different classes in the hierarchy are concerned with separate portions of an object's state. For example, Spec# [LM09] by default prohibits an invariant declared in class `C` from mentioning attributes introduced in its ancestor class `B`; this way `C`'s invariant cannot be violated by the methods it inherited from `B`.[8] Moreover, Spec# has a more fine-grained mechanism for keeping track of object consistency: in place of AutoProof's Boolean attribute `closed`, it features a class-valued attribute called `inv`, which, for an object `x`, ranges from `object` to the dynamic type of `x`, and denotes the most specific class whose invariant must currently hold for `x`. When unwrapping `x`, the programmer specifies the desired new value of `inv`. This fine-grained mechanism allows the methods redefined and newly introduced in class `C` to operate on the new portion of the receiver's state without having to reverify the preservation of the portion of the invariant that originated in class `B`.

We found such prior approaches unsatisfactory for the purpose of verifying EiffelBase2. In EiffelBase2 inheritance is normally used to refine an abstract concept, rather than to extend a concept with an orthogonal aspect, and therefore the invariants of most descendant classes do impose new constraints on inherited state; a prototypical example are linking invariants (see Sect. 3.4). Using Spec#'s approach in this setting would have forced us to mark most of the inherited attributes as *additive* [LW08]—so that they can be mentioned in the descendants' invariants—which in turn would have required us to redefine every method that modifies such additive arguments. In EiffelBase2, however, we can always assume that the source code of an ancestor class is available (Eiffel does not support separate compilation or information hiding within a class hierarchy), and thus methods modifying additive arguments should simply be re-verified without being redefined. Based on these considerations, we chose a different trade-off between conceptual simplicity, annotation overhead, and proof reuse, which is more suitable in our domain.

In AutoProof, method implementations can be declared *covariant* or *nonvariant*. A nonvariant implementation cannot depend on the dynamic type of the receiver object (denoted by `Current`), and hence on the precise definition of its invariant; therefore, a correct nonvariant implementation remains correct in descendant classes with stronger invariants, and need not be re-verified. In contrast, a covariant implementation may depend on the dynamic type of the receiver, and hence must be re-verified when inherited. In practice, method implementations have to be covariant only if they call `Current.wrap`, which is the case for commands that directly modify attributes of `Current` (such as `extend_back` in Fig. 1): `wrap` checks that the invariant holds, a condition that may become stronger along the inheritance hierarchy. Otherwise, queries and commands that modify `Current` indirectly by calling other commands can be declared nonvariant: method `append` in `LINKED_LIST` (not shown in Fig. 1) calls `extend_back` in a loop; it then only needs to know that `Current` is `closed` but not any details of the actual invariant. Nonvariant implementations are a prime example of how decoupling the uninterpreted notion of consistency from invariant definitions (Sect. 3.4) promotes modular verification.

[8] Parkinson and Bierman's support for inheritance in abstract predicate families [PB08] is based on a similar principle, albeit expressed in a more flexible way using separation logic.

3.6.2. *Framing and inheritance*

Another challenge posed by inheritance is defining the semantics of model-based frame specifications (`modify model`), given that models can change with inheritance. When a class `B` inherits from `A` it can: keep `A`'s model attributes unchanged; add new model attributes (to introduce new dimensions to the abstract state); replace some of `A`'s model attributes (to express existing dimensions in a different way); or drop some of `A`'s model attributes (to get rid of dimensions that have become redundant). What is then, within `B`, the semantics of the `modify model` clause of a method `m` inherited from `A`?

Since the clients of a method rely on the heap locations (object/attribute pairs) outside the frame to stay unchanged, soundness implies that the set of heap locations specified by a frame specification can only remain the same or shrink with inheritance. Correspondingly, AutoProof excludes from the frame of method `m` all the model attributes newly introduced in `B`, except those that *replace* model attributes in `A`. Keyword `replace` annotates such new model attributes that replace inherited attributes. For example, a method in `LINKED_LIST` inherited from `CONTAINER` with frame specification `modify model Current` [`bag`] is allowed to modify `sequence` (since it replaces `bag` in `LIST`) but not any other model attribute of `LINKED_LIST`.

3.7. Effective Boogie encoding

The single biggest obstacle to completing the verification of EiffelBase2 has been poor verification performance on large problems: making AutoProof scale required tuning several low-level details of the Boogie translation, following a trial-and-error process. The result is, however, largely domain-independent, as it provides general flexibility to handling class invariants and other kinds of annotations; in fact, AutoProof's performance improved after the tuning also in domains other than data-structure verification, such as algorithmic challenges [TFNP15]. We summarize some finicky features of the translation that turned out to be crucial for performance.

3.7.1. *Invariant reasoning*

Class invariants tend to be the most complex part of specifications in EiffelBase2; thus, their translation must avoid bogging down the prover with too much information at a time. One crucial point is when `x.wrap` is called and all of `x`'s invariant clauses $\texttt{I}_1, \ldots, \texttt{I}_n$ are checked; the naive encoding `assert` $\texttt{I}_1$; ...; `assert` $\texttt{I}_n$ does not work well for complex invariants: for $j < k$, $\texttt{I}_k$ normally does not depend on $\texttt{I}_j$, and hence the previously established fact that $\texttt{I}_j$ holds just clutters the proof space. Instead, we adopt Dafny's calculational proof approach [LP13] and use nondeterministic branching to check each clause independently of the others.

At any program point where the scope includes a closed object `x`, the proof might need to make use of its invariant. AutoProof's default behavior (assume the invariants of all closed objects in scope) doesn't scale to EiffelBase2's complex specifications. Instead, we leverage once again the decoupling between the generic notion of consistency and the specifics of invariant definitions, and make the latter available to the prover selectively, by asserting AutoProof's built-in predicates: `x.inv` refers to `x`'s whole invariant; `x.inv_only(k)` refers to `x`'s invariant clause named `k`; and `x.inv_without(k)` refers to `x`'s invariant without clause named `k` (for example, see `make`'s body in Fig. 1).

3.7.2. *Opaque functions*

Opaque functions are pure functions whose axiomatic definitions can be selectively introduced only when needed.[9] A function `f` declared as opaque is normally uninterpreted; but using a built-in predicate `def(f(args))` introduces `f(args)`'s definition into the proof environment. In EiffelBase2, we use opaque functions to handle some invariant clauses which refer to model components that are inherited but never directly used in the current class. For example, `HASH_TABLE` in Fig. 2 inherits a "legacy" `bag` model from `CONTAINER`; since `HASH_TABLE`'s methods do not reference `bag`, we declare an opaque function that wraps invariant clauses involving `bag` and simply treat it as an uninterpreted predicate within `HASH_TABLE`.

[9] A similar concept was independently developed for Dafny at around the same time [CLS14].

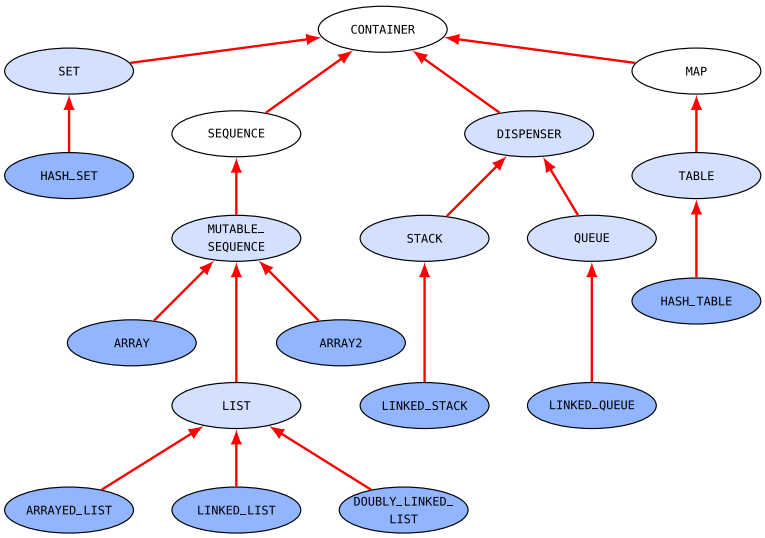

Fig. 5. EiffelBase2 containers: *arrows* denote inheritance; abstract classes have a *lighter background* (*white* for classes with immutable interfaces). (Helper classes are omitted from the diagram for readability)

3.7.3. Modular translation

AutoProof offers the choice of creating a Boogie file per class or per method to be verified. Besides the annotated implementation of the verification module, the file only includes those Boogie theories and specifications that are referenced in the module. We found that minimizing the Boogie input file can significantly impact performance, avoiding fruitless instantiations of superfluous axioms.

4. The verified library

EiffelBase2 was initially designed to replace EiffelBase—Eiffel's standard container library—by providing similar functionalities, a better, more modern design, and assured reliability. It originated as a case study in software development driven by strong interface specifications [PFM10]. Library versions predating our verification effort have been used in introductory programming courses since 2011 (see also Sect. 4.3), and have been distributed with the EiffelStudio compiler since 2012.

4.1. Setup

Pre-verification EiffelBase2 included complete implementations and strong *public* functional specifications. Following the approach described in Sect. 3, we provided additional public specifications for dependent invariants (ownership and collaboration schemes), as well as private specifications for verification (representation invariants, ghost state and updates, loop invariants, intermediate assertions, lemmas, and so on). This effort took about 7 person-months, including extending AutoProof to support special annotations and the efficient encoding of Sect. 3.7. The most time-consuming task—making the tool scale to large examples—was a largely domain-independent, one-time effort; hence, using AutoProof in its present state to verify other similar code bases should require significantly less effort.[10]

Verified EiffelBase2 consists of 46 classes offering an API with 135 public methods; its implementation has over 8000 lines of code and annotations in 79 abstract and 378 concrete methods. The bulk of the classes belong to one of two hierarchies: containers (Fig. 5) and iterators (Fig. 6). Extensive usage of inheritance (including multiple inheritance) makes for uniform abstract APIs and reusable implementations.

[10] We also have some evidence that AutoProof's usability for non-experts improved after extending it as described in this paper. Students in our "Software Verification" course used AutoProof in 2013 (pre EiffelBase2 verification) and in 2014 (post EiffelBase2 verification); working on similar projects, the students in 2014 were able to complete the verification of more advanced features and generally found the tool reasonably stable and responsive [FPT15].

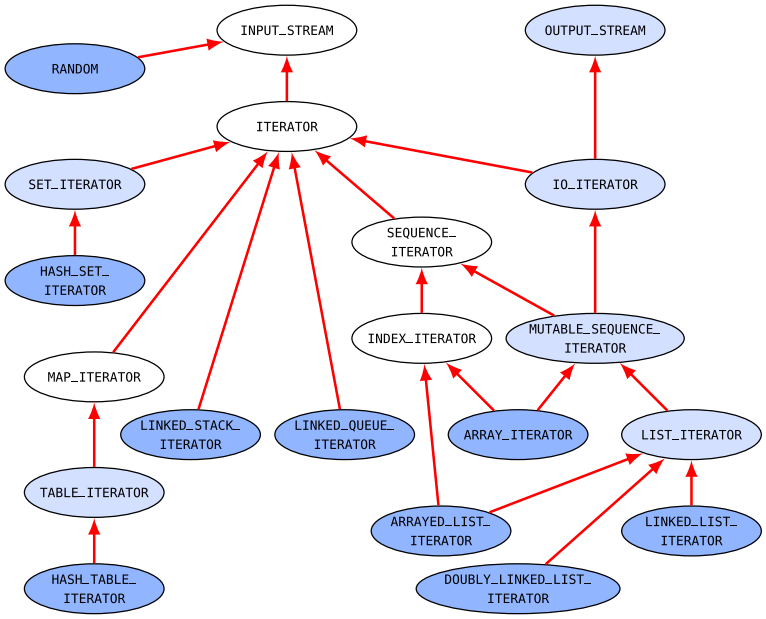

Fig. 6. EiffelBase2 iterators: *arrows* denote inheritance; abstract classes have a *lighter background* (*white* for classes with immutable interfaces). (Helper classes are omitted from the diagram for readability.)

Completeness All but two EiffelBase2 classes have *complete* and *observable* model-based specifications according to the definitions given in Sect. 3.3. The two exceptions represent the two cases of "justified" incompleteness also mentioned in the section: class `RANDOM` features non-deterministic semantics, while class `DISPENSER` factors out the common functionality of stacks and queues, and thus cannot define precisely the semantics of element insertion.

Specifications treat integers as mathematical integers to offer nicer abstractions to clients. Pre-verification EiffelBase2 supports both object-oriented style (abstract classes) and functional style (closures or agents) definitions of object equality and hashing operators; verified EiffelBase2 covers only the object-oriented style. The library has no concurrency-related features: verification assumes sequential execution.

4.2. Verification results

Given the annotations described below, AutoProof verifies all 378 method implementations automatically.

Bugs found A byproduct of verification was exposing 3 subtle bugs in pre-verification EiffelBase2.

- Method `copy_range` in class `MUTABLE_SEQUENCE` failed when copying elements from a range into an overlapping range with a smaller offset inside the same sequence.
- Method `bounded_item` in class `RANDOM` encountered a division by zero as a result of integer conversions between integers of different bit-widths.
- A low-level array service class (wrapping native C arrays) used by EiffelBase2 considered equal any two sub-ranges of the same array.

We attribute the low number of defects in EiffelBase2 to its rigorous, specification-driven development process: designing from the start with complete model-based interface specifications forces developers to carefully consider the abstractions underlying the implementation. An earlier version of EiffelBase2 has been tested automatically against its interface specifications used as test oracles, which revealed 7 bugs [Pol14, Ch. 4] corrected before starting the verification effort. These results confirm the intuition that lightweight formal methods, such as contract-based design and testing, can go a long way towards detecting and preventing software defects; however, full formal verification is still required to get rid of the most subtle few.

Table 1. EiffelBase2 verification statistics: for every CLASS, the number of ABStract and CONCrete methods, and the methods and well-formedness constraints that have to be VERified; the TOTAL number of non-empty non-comment lines of code, broken down into EXECutable code and SPECifications; the latter are further split into REQuirements and AUXiliary annotations; the overhead $\frac{\text{SPEC}}{\text{EXEC}}$ in both LOC (lines) and TOKens; and the verification TIME in seconds: TOTAL time per class, TRANSlation (to Boogie) time per class, and MEDian and MAXimum Boogie running times of the class's methods

CLASS	METHODS ABS	CONC	VER	LOC TOTAL	EXEC	SPEC	REQ	AUX	$\frac{\text{SPEC}}{\text{EXEC}}$	TOK $\frac{\text{SPEC}}{\text{EXEC}}$	TIME (S) TOTAL	TRANS	MED	MAX
CONTAINER	2	3	6	124	39	85	34	51	2.2	3.1	3.5	2.6	0.1	0.3
INPUT_STREAM	3	1	2	59	22	37	32	5	1.7	4.3	2.6	2.0	0.3	0.5
OUTPUT_STREAM	2	2	3	90	34	56	42	14	1.6	4.0	2.9	2.2	0.3	0.3
ITERATOR	12	4	6	241	106	135	106	29	1.3	3.6	3.8	2.6	0.2	0.3
SEQUENCE	4	9	14	182	69	113	102	11	1.6	2.5	4.8	3.0	0.1	0.3
SEQUENCE_ITERATOR	0	1	3	36	15	21	19	2	1.4	2.2	3.1	2.2	0.2	0.4
MUTABLE_SEQUENCE	3	5	8	191	83	108	74	34	1.3	3.5	7.6	3.0	0.2	2.9
IO_ITERATOR	1	1	2	58	15	43	33	10	2.9	4.9	3.1	2.2	0.4	0.5
MUTABLE_SEQUENCE_ITERATOR	1	0	1	41	19	22	11	11	1.2	1.5	2.9	2.2	0.7	0.7
ARRAY	0	16	21	275	149	126	107	19	0.8	1.6	12.5	3.4	0.2	2.1
INDEX_ITERATOR	0	13	14	91	64	27	18	9	0.4	0.3	5.0	2.9	0.1	0.2
ARRAY_ITERATOR	0	3	11	97	43	54	34	20	1.3	2.3	6.2	3.0	0.2	0.6
ARRAYED_LIST	0	20	27	389	196	193	127	66	1.0	1.9	19.5	4.5	0.2	4.3
ARRAYED_LIST_ITERATOR	0	10	18	144	81	63	34	29	0.8	1.2	9.9	3.4	0.3	1.0
ARRAY2	0	16	20	199	101	98	79	19	1.0	1.1	7.4	3.3	0.1	1.0
LIST	11	5	11	268	85	183	129	54	2.2	6.1	6.1	3.1	0.1	1.3
LIST_ITERATOR	7	0	1	118	32	86	86	0	2.7	10.2	3.0	2.3	0.7	0.7
CELL	0	1	3	23	12	11	8	3	0.9	1.1	2.7	2.1	0.1	0.3
LINKABLE	0	1	4	25	14	11	11	0	0.8	0.9	2.8	2.1	0.2	0.2
LINKED_LIST	0	23	30	558	271	287	125	162	1.1	2.1	22.3	4.2	0.3	3.3
LINKED_LIST_ITERATOR	0	28	29	402	205	197	84	113	1.0	2.0	13.6	3.9	0.2	1.4
DOUBLY_LINKABLE	0	5	10	136	37	99	85	14	2.7	3.8	4.2	2.6	0.1	0.8
DOUBLY_LINKED_LIST	0	23	30	641	291	350	147	203	1.2	2.3	31.3	4.3	0.3	10.7
DOUBLY_LINKED_LIST_ITERATOR	0	27	28	379	207	172	66	106	0.8	1.7	13.5	3.9	0.3	1.4
DISPENSER	6	0	3	68	27	41	40	1	1.5	2.9	3.0	2.4	0.1	0.4
STACK	1	0	4	25	12	13	12	1	1.1	2.1	3.2	2.3	0.2	0.3
LINKED_STACK	0	9	12	100	51	49	23	26	1.0	1.5	5.5	3.1	0.2	0.3
LINKED_STACK_ITERATOR	0	16	18	221	94	127	59	68	1.4	2.2	8.6	3.5	0.2	1.0
QUEUE	1	0	4	25	12	13	12	1	1.1	2.0	3.2	2.3	0.2	0.3
LINKED_QUEUE	0	9	12	100	51	49	23	26	1.0	1.5	5.5	3.2	0.2	0.3
LINKED_QUEUE_ITERATOR	0	16	18	221	94	127	59	68	1.4	2.2	8.6	3.5	0.2	1.0
LOCK	0	8	9	176	0	176	176	0			4.2	2.8	0.1	0.6
LOCKER	0	1	2	30	0	30	30	0			2.8	2.1	0.3	0.4
MAP	6	1	8	128	32	96	90	6	3.0	5.0	4.1	3.0	0.1	0.2
MAP_ITERATOR	2	0	4	81	19	62	44	18	3.3	7.4	3.5	2.6	0.2	0.3
TABLE	5	2	5	97	39	58	51	7	1.5	2.4	4.3	2.6	0.3	0.6
TABLE_ITERATOR	2	0	1	43	17	26	26	0	1.5	4.1	3.2	2.4	0.7	0.7
HASHABLE	1	0	1	35	9	26	21	5			2.5	1.9	0.5	0.5
HASH_LOCK	0	2	6	41	0	41	41	0			5.2	2.9	0.2	1.4
HASH_TABLE	0	26	31	695	236	459	208	251	1.9	3.6	61.4	6.5	0.4	8.7
HASH_TABLE_ITERATOR	0	23	29	572	198	374	104	270	1.9	4.1	46.9	5.8	0.8	6.3
SET	7	10	17	503	163	340	217	123	2.1	4.4	28.4	3.3	0.2	11.8
SET_ITERATOR	2	0	2	50	17	33	32	1	1.9	6.2	3.1	2.3	0.4	0.5
HASH_SET	0	10	13	146	59	87	43	44	1.5	1.9	11.1	4.1	0.4	1.1
HASH_SET_ITERATOR	0	17	18	216	91	125	42	83	1.4	2.8	18.8	4.5	0.6	2.7
RANDOM	0	11	12	100	78	22	21	1	0.3	0.3	3.4	2.6	0.1	0.1
Total	**79**	**378**	**531**	**8440**	**3489**	**4951**	**2967**	**1984**	**1.4**	**2.7**	**434.7**	**140.9**	**0.2**	**11.8**

4.2.1. Specification Succinctness

Table 1 details the size of EiffelBase2's specifications: overall, 1.4 lines of annotations per line of executable code. The same overhead in tokens—generally considered a more robust measure—is 2.7 tokens of annotation per token of executable code. The overhead is not uniform across classes: abstract classes tend to accumulate a lot of annotations which are then amortized over multiple implementations of the same abstract specification.

EiffelBase2's overhead compares favorably to the state of the art in full functional verification of heap-based data-structure implementations. Pek et al's [PQM14] verified list implementations have overheads of 0.6 (LOC)

and 2.6 (tokens);[11] given that their technique specifically targets inferring low-level annotations, EiffelBase2's specifications are generally succinct. In fact, approaches without inference or complete automation tend to require significantly more verbose annotations. Zee et al.'s [ZKR08] linked structures have overheads of 2.3 (LOC) and 8.2 (tokens); their interactive proof scripts aggravate the annotation burden. Java's `ArrayList` verified with separation logic and VeriFast [Ver16] has overheads of 4.4 (LOC) and 10.1 (tokens).

4.2.2. Kinds of Specifications

Pek et al. [PQM14] suggest classifying specifications according to their level of abstraction with respect to the underlying verification process. A natural classification for EiffelBase2 specifications is into *requirements* (model attributes, method pre/post/frame specifications, class invariants, and ghost functions directly used by them) and *auxiliary* annotations (loop invariants and variants, intermediate assertions, lemmas, and ghost code not directly used in requirements). Requirements are higher level in that they must be provided independent of the verification methodology, whereas auxiliary annotations are a pure burden which could be reduced by inference. EiffelBase2 includes 3 lines of requirements for every 2 lines of auxiliary annotations. In terms of API specification, clients have to deal with 6 invariant clauses per class and 4 pre/post/frame clauses per method on average.

Auxiliary annotations can be further split into *suggestions* (`inv`, `inv_only`, and `inv_without` and opaque functions, all described in Sect. 3.7) and *structural* annotations (all other auxiliary annotations). Suggestions roughly correspond to Pek et al.'s "level-C annotations" [PQM14], in that they are hints to help AutoProof verify more quickly. Out of all EiffelBase2 specifications, 12% are suggestions (mostly `inv` assertions); among structural annotations, ghost code (11%) and loop invariants (7%) are the most significant kinds. The 3/2 requirements to auxiliary annotation ratio indicates that high-level specifications prevail in EiffelBase2. The non-negligible fraction of auxiliary annotations motivates future work to automatically infer them (in particular, suggestions) when possible.

4.2.3. Default annotations

Default annotations help curb the annotation overhead. Default wrapping calls (Sect. 2.1) work for 83% of method bodies, and default `closed` pre-/postconditions work for 95% of method specifications; we overrode the default in the remaining cases. Implicit ghost attribute updates (Sect. 3.2) always work.

4.2.4. Verification performance

In our experiments, 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. To account for noise, we ran each verification 30 times and report the mean of the values in the 95th percentile.[12]

The total verification time is under 8 minutes, during which AutoProof verified 531 method implementations and well-formedness conditions, including the 378 concrete methods listed in Table 1, 47 ghost methods and lemmas, well-formedness of each class invariant, and 56 inherited methods that are covariant (Sect. 3.6) and hence must be re-verified; on the other hand, nonvariant annotations avoid re-verification of 343 bodies, which saved about 30% of the total verification time.

AutoProof's behavior is not only well-performing on the whole EiffelBase2; it is also *predictable*: over 99% of the methods verify in under 10 s; over 89% in under 1 s; the most complex method verifies in under 12 s. 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; for example, independent checking of invariant clauses (Sect. 3.7) halves the verification time of some of the most complex methods. Steady, predictable performance is key to making AutoProof a mature tool that is robustly applicable.

4.3. Client Reasoning

While the focus of this paper is describing how to specify and verify a library, EiffelBase2's model-based interface specifications should also support reasoning about its *clients*.

[11] We counted specifications used by multiple procedures only once.

[12] That is, we discard outliers that are above the 95% percentile—the value below which 95% of the recorded times are.

Table 2. Verifying client code using EiffelBase2 (content from [FPT15, Tab. 2])

PROGRAM		CLASSES	METHODS	LOC TOTAL	LOC EXEC	LOC SPEC	LOC REQ	LOC AUX	$\frac{\text{SPEC}}{\text{EXEC}}$	TOK $\frac{\text{SPEC}}{\text{EXEC}}$	METHODS VERIFIED %
	Before	4	9	184	164	20	20	0	0.1	0.2	
Board Game 1	After	4	8	266	165	101	55	46	0.6	1.2	100
	Before	8	19	342	301	41	41	0	0.1	0.3	
Board Game 2	After	8	18	490	307	183	108	75	0.6	1.4	100
	Before	15	38	578	491	87	87	0	0.2	0.2	
Board Game 3	After	15	45	1033	608	425	253	172	0.7	1.1	93
	Before	13	145	1661	1219	442	442	0	0.4	0.5	
Traffic Library	After	14	149	2297	1220	1077	811	266	0.9	1.3	78
	Before	40	211	2765	2175	590	590	0	0.3	0.4	
Total	After	41	220	4021	2300	1786	1227	559	0.8	1.2	84

Each PROGRAM occupies two consecutive rows: the first refers to the program *before* introducing the annotations for verification, the second to the program *after* introducing all annotations. Each row lists the number of CLASSES the program consists of, and the number of METHODS in those classes (which may vary between *before* and *after* due to refactoring); the TOTAL number of non-empty non-comment lines of code, broken down into EXECutable code and SPECifications; the latter are further split into REQuirements and AUXiliary annotations; the overhead $\frac{\text{SPEC}}{\text{EXEC}}$ in both LOC (lines) and TOKens; and the percentage of VERIFIED METHODS

Client-side verification encompasses challenges that are somewhat different from those involved in verifying a library. In particular, client code may have a design less conducive to verification because it lacks well-defined interfaces, so that even accurately specifying the expected behavior turns out to be cumbersome. To demonstrate how EiffelBase2's interface specifications enable client reasoning, we verified parts of four programs that are clients of EiffelBase2.

In contrast to the library verification effort, which aimed at full functional correctness, our *goal* in this client-verification experiment was to find out how much verification is achievable in a setting somewhat representative of serious non-expert (library) users, namely 5 person-days spent on verification with only minimal modifications to the preexisting client code—and no modifications at all to AutoProof—allowed. The rest of this section summarizes this experience and highlights some open challenges; in a related publication [FPT15] we reflect in greater detail on AutoProof's usability to verify pre-existing client code.

4.3.1. Four Client Programs of EiffelBase2

As part of the exercise sessions of ETH's *Introduction to Programming* course for first-year computer science majors, students had to develop three applications (board games with increasingly richer features inspired by Monopoly) and extend one library (modeling a transportation system) with new features. Here is a brief description of the four programs (the three applications and the library).

Board Game 1 Players roll dice to advance on a board of fixed size. A player's state consists of a name and a position on the board.

Board Game 2 Extending Board Game 1, players collect money according to the positions on the board they visit. Consequently, a player's state also includes an amount of collected money. Each cell on the board belongs to one of three categories (win money, lose money, no money change), represented by classes related by inheritance.

Board Game 3 Extending Board Game 2, board cells support more complex behavior such as properties that can be built by one player and that force other visiting players to pay rent.

Traffic Library Modeling entities of an urban public transportation network, such as lines, stops, and carriers. Traffic's classes make extensive usage of mutually-dependent invariants to define consistency between the content of their data structures.

As one can appreciate even from these curt descriptions, all four programs use structures, such as arrays, lists, streams, tables, and random number generators, which are provided by EiffelBase2. We took the master solutions of the four programming assignments, written by instructors in the past using pre-verification EiffelBase2 (see Sect. 4.1); we refer to the master solutions as "before" programs—as in "before verification". The before programs are complete executable implementations annotated with some (usually incomplete) model-based interface specification in the form of preconditions, postconditions, and class invariants.

4.3.2. Verifying Client Code

Our goal was verifying the before programs against their interface specifications and verified EiffelBase2, which subsumes proving correctness of the interactions between EiffelBase2 and the programs (for example, iterator safety). This required adding annotations for verification, including frame specifications, specifications of private methods, and ghost code. We tried, as much as possible, to retain the interface specifications and the implementations given in the before programs. When this was not possible—because the before programs were written without verification in mind—we limited ourselves to minimal changes that did not affect the programs' overall interface, structure, or input/output behavior. The result of this effort are the "after" programs, which can be verified with AutoProof. Table 2 gives some statistics about size, specification, and verification of the four programs both in their before and in their after versions.

Annotations and unsupported features We added annotations mainly to: strengthen interface specification to support modular reasoning; provide frame specifications; specify object dependencies according to the ownership and semantic collaboration schemes (Sect. 3); reason about loops using invariants and variants; provide intermediate goals to AutoProof. Two features of the Eiffel language, not fully supported by AutoProof, required some extra legwork: to reason about strings, we annotated with model-based contracts a custom, simplified version of system class `STRING`; to reason about `once` methods (Eiffel's twist on static methods), we made their stateful semantics explicit by introducing additional attributes.

Client verification results Obviously, the success of verification depended on the complexity of the programs and specifications. The after programs total 41 classes and 4086 lines of code. Within 5 person-days we produced 1321 lines of annotations as well as minor modifications to the code. Verification of 84% of 220 methods succeeded; the exceptions were in large classes that use up to 7 complex data structures simultaneously, where accumulated specification complexity bogs down AutoProof; verifying these complex parts would require restructuring the code to improve its modularity. Relative to the budget that we allowed ourselves for this experiment, the results are overall satisfactory and suggest that EiffelBase 2's interfaces naturally enable client verification.

Board Game 1 and 2 Verification was completely successful with reasonable effort and only minimal changes to the implementation.[13] Providing loop invariants was the most time consuming task; as usual the loop invariants had not only to be correct but also to be amenable to automated reasoning without bogging down the prover. To reason about complex class invariants we relied on the same technique of only introducing invariants in the proof context by request that we added to AutoProof for EiffelBase2's verification (see Sect. 3.7.1). This made it possible to handle complex class invariants involving multiple objects and classes, but also required more custom annotations to select which facts are relevant to each proof context.

Board Game 3 Verification was successful for most but not all (93%) of the methods, and required more widespread changes to the implementation compared to Board Game 1 and 2. We could not satisfactorily specify the framing specification of some features involving multiple objects in class `PROPERTY`. This class models the buildings that occupy the location of a board's cell and the related notion that they are a player's property—other players have to pay when they visit it. Specifying the relations between the three entities (location, building, and players) is not possible at the level of `PROPERTY`'s predecessor class, which models a simpler kind of cell that has no notion of property. Specifying the relations at the level of class `PROPERTY` itself is also not possible fully, because it contradicts part of the specification inherited from `PROPERTY`'s predecessor. Ultimately, the design of `PROPERTY` is not fully consistent, and this prevented us from completely verifying it without reworking its implementation. Another aspect that required some special care was the verification of constructors; we introduced code to handle the initialization of ghost attributes related to ownership and collaboration in a suitable way.

Traffic Library Verification was often successful (78% of the methods), but sometimes failed to scale due to Traffic's highly-coupled design. Traffic's design is an exercise in object-oriented modeling with little concern for modular verification. A core class in the library includes as attributes several data structures holding all entities in the transportation network, and expresses their mutual consistency with a large, complex class invariant. As a result, even if the methods in Traffic are not substantially larger than the methods in Board Game 3, verification tends to be slower and to require quite a bit of explicit ghost-code manipulation to

[13] The verified after code of Board Game 1 and 2 is available at http://tiny.cc/autoproof-repo.

reason about the quantified invariants in several dependent data structures. Floating point arithmetic is one feature used in Traffic (to compute distances between stops) that AutoProof does not fully support: AutoProof translates floating point types to Boogie's `real` type, which represents mathematical reals and is not fully supported by Boogie anyway at the time of writing [AF16]. Hence, we weakened the postconditions of Traffic's methods involving floating point arithmetic to match AutoProof's limited reasoning capabilities in this domain.

4.4. Challenges

After presenting EiffelBase2's verification results, we are well poised to understand the features of a realistic, general-purpose library that presented novel challenges to verification

Section 2 outlined the challenges behind *safe iterators*, which can read and modify container content with multiple iterators active on the same structure, and *mutable hash keys*. EiffelBase2's iterators are safe in that client code verified against their specification won't access containers using invalid iterators. Hash containers provide safe mutable keys with succinct abstract specifications whose usage is also safe.

4.4.1. General-purpose APIs

To be general-purpose, EiffelBase2 offers feature-rich public interfaces, which amplify verification complexity. For example, lists support searching, inserting and removing elements, and merging container's content, at arbitrary positions, replacing and removing elements by value, reversing in place. Sets provide operations for subset, join, meet, (symmetric) difference, and disjointness check. All EiffelBase2's containers also offer copy constructors and object comparison—standard features in object-oriented design but far from trivial to prove correct and routinely evaded in verification.

4.4.2. Object-oriented design

Abstract classes provide uniform, general interfaces to clients, and to this end are extensively used in EiffelBase2, but also complicate verification in different ways. First, the generality of abstract specifications may determine a wider gap between specification and implementation than if we defined specifications to individually fit each concrete implementation. For example, `ITERATOR.forth`'s precondition `all s ∈ subjects : s.closed` involves a quantification that could be avoided by replacing it with the equivalent `target.closed`. However, the quantified precondition is inherited from `INPUT_STREAM`, where `target` is not yet defined. Second, model attributes may be refined with inheritance, which requires extra invariant clauses to connect the new and the inherited specifications (see the discussion about `seq_refines_bag` in Sect. 3.2).

4.4.3. Realistic implementations

Implementations in EiffelBase2 offer realistic performance, in line with standard container libraries in terms of running time and memory usage, which adds algorithmic verification complexity atop structural verification complexity. For example, `ARRAYED_LIST`'s implementation uses, like C++ STL's `Vector`, a ring buffer to offer efficient insertions and deletions at both list ends. Ring buffers were a verification challenge in a recent competition [FPS12]; EiffelBase2's ring buffers are even more complicated as they have to support insertions and deletions inside a list, which requires a circular copy. Another example is `HASH_TABLE`, which implements transparent resizing of the bucket array to maintain a near-optimal load factor—one more feature of realistic libraries that is normally ignored in verification work.

5. Related work

Thanks to their well-defined interface behavior and tricky implementations, data structures are a common source of complex examples where verification systems prove their mettle. We discuss verification techniques that have been applied to significant data-structure implementations, and highlight differences with our work in scope or technical aspects; the presentation roughly proceeds towards approaches that are closer to ours. For brevity, we do not consider unsound approaches, such as those based on finitization or testing.

5.1. Verification of data-structure clients

Properly engineered data-structure components export well-defined abstract interfaces, which makes the problem of verifying safety of *client* code somewhat simpler than verifying the components' implementations themselves against their specification: clients normally access data structures through a limited number of iteration patterns, whose semantics can be formalized and reasoned upon.

Gregor and Schupp [GS06] perform static checking (based on symbolic execution) of C++'s Standard Template Library (STL) containers. The basic idea is dealing with STL's idiomatic iterator features as if they were language extensions, whose semantics is hard-coded as symbolic execution constraints. This approach supports reasoning about validity of iterators (e.g., if a list is modified, its active iterators become invalid) but does not capture full functional correctness; the focus is on error finding, demonstrated on STL's test suite.

Blanc et al. [BGK07] also target C++ STL client code, using predicate abstraction and model checking. They axiomatize the interface behavior of STL containers, and build an operational model (using arrays to represent data) that can be model checked against client code. The axiomatization does not cover functional correctness but safety properties such as memory safety; properties that involve universal quantification may determine spurious counterexamples.

Dross et al. [DFM11] describe a detailed axiomatization of lists, sets, and tables in the Why functional language, as well as an operational semantics in the Coq interactive prover [Pau11]; using Coq, they prove that the Why axiomatization is consistent with the Coq formalization. The axiomatization is designed to be amenable to automated reasoning using SMT solvers; hence it is applicable to proving properties of code using containers written in Ada 2012—even though the paper does not report about concrete case studies.

Dillig et al. [DDA11] use a sound abstract interpretation to handle expressive specifications of STL containers in client code. After annotating the containers' interfaces, they verify memory safety of real C++ applications such as Inkscape, and functional correctness of 15 small examples such as copying a vector and reversing a mapping.

5.2. Verification of individual data structures

Implementing and verifying individual data structures is a quite different exercise than targeting general-purpose data-structure collections. Individual implementations tend to focus on aspects that are amenable to reasoning using the tool or methodology at hand, and that demonstrate the most fundamental aspects of a verification challenge; but they abstract away other details, which often are crucial in a realistic and general-purpose implementation such as EiffelBase2.

Cok [Cok06], and Jacobs et al. [JPS06] introduce preliminary ideas about specifying and checking iterators, respectively in Java and Spec#.

Lahiri and Qadeer [LQ08] focus on SMT-friendly logic fragments suitable to express constraints involving quantification on interpreted sets. They demonstrate their technique on various algorithms operating on linked lists; since it's edging on undecidability, it is unlikely that their technique generalizes to more complex data structures and properties.

The gallery of examples verified using Why3 [Why16] includes individual challenging data structures such as sparse arrays, binary heaps, array-based circular queues, hash tables (with both open and closed hashing), AVL and snapshotable trees, as well as algorithms operating on arrays, lists, and red-black trees. The Why3 language is a functional language (a dialect of ML) specifically designed for verification at a high level of abstraction and with a focus on algorithmic challenges. The Pangolin functional programming language has a similar focus; Régis-Gianas and Pottier [RP08] used the Pangolin verification system to prove the correctness of purely functional implementations of balanced binary search trees and double-ended queues.

VeriFast [Ver16] is a separation-logic tool for Java and C, which has been used to verify iterators and stacks in Java, and generic containers, concurrent linked sets, and lock-free queues in C. The examples are written specifically for verification, and normally cover full functional correctness. Separation logic tools such as VeriFast may require a heavier annotation burden that includes low-level directives to "bundle/open" and "unbundle/close" abstract predicate definitions in crucial parts of a proof. To provide automation at a higher level, Piskac et al. [PWZ14] combine separation logic with decidable first-order fragments to verify programs in a C-like fragment with native support for specification constructs. They demonstrate their GRASShoper verifier on flat linked data-structure examples; it currently cannot handle nested structures or arrays.

Interactive provers have also been applied to the verification of individual data structures. Gamboa [Gam09] discusses some data structures verified in ACL2; the tool's functional language provides high-level representations.

Mehnert et al. [MSBS12] present the first full functional verification of snapshotable trees (a heap-based data structure with complex update patterns) using the Coq interactive prover and annotations in separation logic. Using a similar approach (but also manual proofs on paper), Jensen et al. [JBS11] verify linked lists with views. Bruns [Bru11], and Gladisch and Tyszberowicz [GT13] use the KeY system to verify various implementations of linked lists and red-black trees. Their solution [GT13] addresses some challenges of verification: using specification queries instead of ghost state, and improving the level of automation (KeY offers a combination of interactive and automated back-end provers).

Dafny is an auto-active verifier and language specifically designed for functional correctness proofs. Dafny's examples [Daf16] include individual implementations of challenging data structures: binary trees, lazy, sparse, and extensible arrays, recursively defined lists, (priority) queues, stacks, snapshotable trees, cached containers, streams and tree streams, list iterators. Leino and Moskal [LM10b] describe the challenges behind verifying some of these structures. Like other languages designed specifically for verification, Dafny is flexible and expressive enough to abstractly capture essential aspects of different implementation and specification styles—if at the expense of simple client reasoning; for instance, one of the Dafny examples [Daf16] consists of linked lists with predicates in separation logic style, which Dafny doesn't natively support but can encode. On the flip side, several aspects are abstracted away that are crucial in realistic implementations.

5.3. Verification of data-structure collections

We now move towards verification efforts that deploy the same techniques on multiple data structures at once.

5.3.1. Functional programming languages

Functional languages provide a higher level of abstraction than heap-based (object-oriented) ones, and their powerful type systems can naturally capture nontrivial correctness properties. Therefore, verifying data structures implemented in functional languages poses challenges largely different from those of the implementations we target in this paper.

Refinement approaches Refinement techniques provide a very different approach to creating verified data structures implementations: design and verify a high-level model of the structures, and then refine it into a provably correct executable implementation. Hawkins et al. [HAF+11] do this from high-level relational specifications of data-structure behavior. In order to have implementations with predictable performance, they provide written implementations of basic data structures by wrapping components of C++'s STL; the refinement algorithm combines these elementary components into more complicated structures that satisfy the specifications by construction. Lochbihler [Loc13] first constructs interactive proofs of high-level functional data-structure definitions in Isabelle; and then refines them into executable code. The experiments focus on exploring different provably correct refinements of sets and maps.

Functional verification Xi and Pfenning [XP99] introduce indexed types, whose constraints can capture functional correctness requirements (e.g., sortedness) of recursively defined data structures. Indexed types are limited in the kinds of functional correctness properties they can express, and require a high annotation overhead since no general type inference algorithm has been suggested. To provide for more automation and expressiveness, the Liquid Haskell project introduces recursive and polymorphic type refinements, two mechanisms that can naturally capture functional correctness invariant properties of data structures defined in functional fashion. Vazou et al. [VSJ14] apply these techniques to verify termination, memory safety, and selected functional properties of over 10,000 lines of Haskell code from realistic libraries. Kawaguchi et al. [KRJ09] apply similar techniques to verify ML implementations of lists, vectors, maps, and trees. As they target quantifier-free constraints, Liquid Haskell's techniques are completely automatic: whenever they are applicable, type inference algorithms supply

all the intermediate type annotations (including invariants). On the other hand, these techniques are inapplicable to heap-based data structures; and insisting on full decidability entails that some complex properties cannot be expressed and reasoned upon.

5.3.2. Heap-based programming languages

If we consider realistic data-structure implementations in heap-based programming languages, a large number of works have applied fully automated techniques to verify *simple* properties such as memory safety or limited functional properties. In contrast, targeting full functional correctness requires to give up full automation or to provide additional annotations. This section describes both kinds of works.

Fully automatic verification of simple properties The Code Contracts static checker (formerly, Clousot) is based on abstract interpretation; it has been applied to .NET's standard libraries, which include generic collections, to check the absence of errors such as out-of-bounds array accesses, null dereferences, buffer overruns, and division by zero [LL11], as well as simple properties of arrays [CCL11]. Both works [LL11, CCL11] target precise automatic inference and safety checks, not verification of functional correctness.

Shape analysis aims at reachability properties of objects in the heap. Such shape properties are often accessory to proving full functional correctness; for example, a well-formed linked list must be acyclic. Beyer et al. [BHT06, BHTZ10] perform shape analysis using model-checking techniques based on refinement of predicate abstractions. Sagiv et al. [SRW02] introduce an abstract interpretation-based technique for precise shape analysis. Yang et al. [YLB+08], and Calcagno et al. [CDOY11] use separation logic techniques to reason automatically about shape properties. Gulwani et al. [GMT08], and Itzhaky et al. [IBR+14] extend shape analysis techniques to support bounded quantification over abstract predicates; this way, they can verify some functional properties of linked structures (for example, that a sorting algorithm returns a sorted list) on top of reachability. Within the limits of the properties they can express, all these analysis techniques are applicable to realistic data-structure implementations in real programming languages.

Decidable expressive data-structure abstractions Approaches that gradually extend expressive first-order theories without encroaching into undecidable territory can provide effective abstractions to reason about functional properties fully automatically. A series of works by Kuncak et al. [KPS10, KPSW10, WMK11, JK11, SSK11, WMK12] have focused on decidable theories of sets and bags, which capture essential traits of the interface behavior of data structures, but cannot express exactly the semantics of more complex operations (especially in heap-based languages that may access elements in any order). Chin et al. [CDNQ12] combine shape and functional properties by means of separation logic predicates capturing abstractions such as sets and bags. They demonstrate their work on linked data structures (singly and doubly linked lists, AVL and red-black trees, and priority queues); some of their examples do not cover full functional correctness, and their technique "cannot handle map, sequence, or nonlinear properties".

Interactive verification Nanevski et al. [NMS+08] extend Coq's type system with monadic types that incorporate Hoare-style pre/postcondition specifications of effectful behavior. Such types can encode the semantics of heap-modifying programs within a purely functional language. They demonstrate their technique by verifying association lists, hash tables, and splay trees, as well as map-like iterators. Some aspects of their implementations are still at variance with how general-purpose data structures are implemented in practice; thanks to the functional framework, they belong to an abstraction level that can be considered higher than EiffelBase2's. Verification uses the Coq interactive prover, with significant overhead in terms of required proof scripts. Continuing the same line of work, Chlipala et al. [CMM+09] improve on this aspect by providing partial automation. They focus on reasoning in higher-order separation logic about sharing and aliasing of programs featuring a mix of functional (e.g., higher-order functions) and effectful (e.g., pointer structures) constructs. They demonstrate their approach on association lists, linked lists, stacks, queues, hash tables, and trees. A distinctive perk of this approach (in both works [NMS+08, CMM+09]) is that it can verify the correctness of the verification condition generation process itself—something hardly ever considered in other verification approaches—leading to fully trusted implementations.

Auto-active verification tries to provide a high degree of automation, but without sacrificing the expressiveness needed for full functional correctness. In this line of work, Zee et al. [ZKR08] document a landmark result in verifying full functional correctness of a significant collection of complex data structures. Their Jahob system

provides a high degree of automation by combining provers for various decidable fragments; however, discharging the most complex verification conditions still requires interactive proofs, which make their annotation overhead much higher than ours. Another major difference with our work is that Zee et al. do not always consider general-purpose implementations (for example, hash tables only offer reference-based key comparison, which is too limiting in practice), nor do they target a unitarily designed library. Pek et al.'s natural proofs [PQM14] do not require proof scripts and drastically reduce the annotation burden by inferring auxiliary (low-level) annotations; the resulting annotation overhead is slightly lower than ours (Sect. 4.2). Their technique is based on expressive separation logic specifications of C programs and works on top of VCC. They demonstrate their VCDryad tool on complex data structures including singly and doubly linked lists, and various kinds of trees; some implementations are taken from real C programs (`glib` and OpenBSD). Compared to EiffelBase2, their examples consist of a self-contained individual program for each functionality, and hence do not represent aspects of container libraries with uniform interfaces that contribute to verification complexity. Another difference with our work is that Pek et al. do not always prove full functional correctness; reversal and sorting of linked lists, for example, only verify that the sets of elements are not altered but ignore their order.

In Sect. 4.4, we discussed the challenges specific to verifying a general purpose data-structure library such as EiffelBase2; addressing those challenges is one of the main contribution of our work and how it improves over the state of the art we discussed in this section.

6. Lessons learned and conclusions

We offer as conclusions the main insights into verifying realistic software and building practical verification tools that emerged from our work.

Auto-active verification demands predictability Usable auto-active verification requires predictable, moderate response time to keep users engaged in successive iterations of the feedback loop. We found timeouts a major impediment, wasting time and providing completely uninformative feedback; others report similar experiences [CLS14]. The primary source of timeouts were futile instantiations of quantified axioms; the solution involved profiling the SMT solver's behavior and designing effective triggers. This effort paid off as it made AutoProof's performance quite stable. However, constructing efficient axiomatizations for SMT solvers remains somewhat of a black art; automating this task is an attractive direction for future research.

Realistic verification calls for flexible tools Verifying EiffelBase2 required a combination of effective predefined schemas (to avoid verbose, repetitive annotations of myriad run-of-the-mill cases) and full control (to tackle the challenging, idiosyncratic cases); as a result, AutoProof includes a lot of control knobs with useful defaults. This determines a different trade off than tools (such as Dafny and VeriFast) implementing bare-bones pristine methodologies, which are easier to learn but offer less support to advanced users that go the distance.

Verification promotes good design It's unsurprising that well-designed software is easier to verify; the flip-side is that developing software with verification in mind is conducive to good design. Verification commands avoiding any unnecessary complexity—a rigor which can pay off manyfold by leading to better reusability and maintainability.

It remains that the vision of "developers of data structure libraries [delivering] formally specified and fully verified implementations" [ZKR08] is still ahead of us. An important step towards achieving this vision, our work explored the major hurdles that lie in the often neglected "last mile" of verification—from challenging benchmarks to fully-specified general-purpose realistic programs—and described practical solutions to overcome them.

References

[AF16] Ameri M, Furia CA (June 2016) Why just Boogie? Translating between intermediate verification languages. In: Proceedings of the 12th international conference on integrated formal methods (iFM), volume 9681 of lecture notes in computer science. Springer, pp 1–17

[BCD+05] Barnett M, Chang B-YE, DeLine R, Jacobs B, Leino KRM (2005) Boogie: a modular reusable verifier for object-oriented programs. In: FMCO, pp 364–387

[BDF+04] Barnett M, DeLine R, Fähndrich M, Leino KRM, Schulte W (2004) Verification of object-oriented programs with invariants. J Object Technol 3(6):27–56

[BGK07] Blanc N, Groce A, Kroening D (2007) Verifying C++ with STL containers via predicate abstraction. In: 22nd IEEE/ACM international conference on automated software engineering (ASE 2007), Nov 5–9, 2007, Atlanta, Georgia, USA, pp 521–524

[BHT06] Beyer D, Henzinger TA, Théoduloz G (2006) Lazy shape analysis. In: 18th international conference on computer aided verification, CAV 2006, Seattle, WA, USA, Aug 17–20, 2006, Proceedings, volume 4144 of lecture notes in computer science. Springer, pp 532–546

[BHTZ10] Beyer D, Henzinger TA, Théoduloz G, Zufferey D (2010) Shape refinement through explicit heap analysis. In: Fundamental approaches to software engineering, volume 6013 of lecture notes in computer science. Springer, pp 263–277

[BN04] Barnett M, Naumann DA (2004) Friends need a bit more: maintaining invariants over shared state. In: 7th international conference on mathematics of program construction, MPC 2004, Stirling, Scotland, UK, July 12–14, 2004, Proceedings, pp 54–84

[Bru11] Bruns D (2011) Specification of red-black trees: showcasing dynamic frames, model fields and sequences. In: 10th keY symposium, Nijmegen, The Netherlands, Extended Abstract.

[CCL11] Cousot P, Cousot R, Logozzo F (2011) A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, Austin, TX, USA, Jan 26–28, 2011. ACM, pp 105–118

[CDH+09] Cohen E, Dahlweid M, Hillebrand MA, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S (2009) VCC: a practical system for verifying concurrent C. In: 22nd international conference on theorem proving in higher order logics, TPHOLs 2009, Munich, Germany, Aug 17–20, 2009. Proceedings, volume 5674 of lecture notes in computer science. Springer, pp 23–42

[CDNQ12] Chin W, David C, Nguyen HH, Qin S (2012) Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci Comput Program 77(9):1006–1036

[CDOY11] Calcagno C, Distefano D, O'Hearn PW, Yang H (2011) Compositional shape analysis by means of bi-abduction. J ACM 58(6):26

[Cha06] Charles J (2006) Adding native specifications to JML. In: Workshop on formal techniques for java-like programs (FTfJP)

[CLRS09] Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. The MIT Press, Cambridge

[CLS14] Christakis M, Leino KRM, Schulte W (2014) Formalizing and verifying a modern build language. In: FM 2014: formal methods—19th international symposium, Singapore, May 12–16, 2014. Proceedings, volume 8442 of lecture notes in computer science. Springer, pp 643–657

[CLSE05] Cheon Y, Leavens G, Sitaraman M, Edwards S (2005) Model variables: cleanly supporting abstraction in design by contract. Softw Pract Exper 35(6):583–599

[CMM+09] Chlipala A, Malecha JG, Morrisett G, Shinnar A, Wisnesky R (2009) Effective interactive proofs for higher-order imperative programs. In: Proceeding of the 14th ACM SIGPLAN international conference on functional programming, ICFP 2009, Edinburgh, Scotland, UK, Aug 31–Sept 2, 2009. ACM, pp 79–90

[Cok06] Cok DR (2006) Specifying Java iterators with JML and ESC/Java2. In: Proceedings of the 2006 conference on specification and verification of component-based systems, SAVCBS '06. ACM, pp 71–74

[Daf16] Dafny example gallery. http://dafny.codeplex.com/SourceControl/latest. Last access Feb 2016.

[DB82] Dunlop DD Basili VR (1982) A comparative analysis of functional correctness. ACM Comput Surv 14(2):229–244

[DDA11] Dillig I, Dillig T, Aiken A (2011) Precise reasoning for programs using containers. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'11, New York, NY, USA. ACM, pp 187–200

[DFM11] Dross C, Filliâtre J-C, Moy Y (2011) Correct code containing containers. In: 5th international conference on tests and proofs (TAP'11), volume 6706 of lecture notes in computer science, Zurich. Springer, pp 102–118

[dMB08] Moura dL Bjørner N (2008) Z3: an efficient SMT solver. In:14th international conference tools and algorithms for the construction and analysis of systems, TACAS 2008, held as part of the Joint European conferences on theory and practice of software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings, volume 4963 of lecture notes in computer science. Springer, pp 337–340

[FGP14] Filliâtre J, Gondelman L, Paskevich A (2014) The spirit of ghost code. In: Proceedings of the 26th international conference on computer aided verification (CAV), volume 8559 of lecture notes in computer science. Springer, pp 1–16

[FNPT16] Furia CA, Nordio M, Polikarpova N, Tschannen J (2016) AutoProof: auto-active functional verification of object-oriented programs. Int J Softw Tools Technol Transf, Online since April 2016. http://link.springer.com/article/10.1007/s10009-016-0419-0.

[FPS12] Filliâtre J-C, Paskevich A, Stump A (2012) The 2nd verified software competition: experience report. In: COMPARE, volume 873 of CEUR workshop proceedings. CEUR-WS.org, https://sites.google.com/site/vstte2012/compet.

[FPT15] Furia CA, Poskitt CM, Tschannen J (June 2015) The AutoProof verifier: Usability by non-experts and on standard code. In: Proceedings of the 2nd workshop on formal integrated development environment (F-IDE), volume 187 of electronic proceedings in theoretical computer science. EPTCS, Workshop co-located with FM 2015, pp 42–55

[Gam09] Gamboa RA (2009) A formalization of powerlist algebra in ACL2. J Autom Reason 43(2):139–172

[GMT08] Gulwani S, McCloskey B, Tiwari A (2008) Lifting abstract interpreters to quantified logical domains. In: Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2008, San Francisco, California, USA, Jan 7–12, 2008. ACM, pp 235–246

[GS06] Gregor D, Schupp S (2006) STLlint: lifting static checking from languages to libraries. Softw Pract Exper 36(3):225–254

[GT13] Gladisch C Tyszberowicz S (2013) Specifying a linked data structure in JML for formal verification and runtime checking. In: Brazilian symposium on formal methods (SBMF), volume 8195 of lecture notes in computer science. Springer, pp 99–114

[HAF+11] Hawkins P, Aiken A, Fisher K, Rinard M, Sagiv M (2011) Data representation synthesis. In: Proceedings of the 32Nd ACM SIGPLAN conference on programming language design and implementation, PLDI'11, New York, NY, USA. ACM, pp 38–49

[HLL+12] Hatcliff J, Leavens GT, Leino KRM, Müller P, Parkinson MJ (2012) Behavioral interface specification languages. ACM Comput Surv 44(3):16

[IBR+14] Itzhaky S, Bjørner N, Reps TW, Sagiv M, Thakur AV (2014) Property-directed shape analysis. In: 26th international conference computer aided verification, CAV 2014, Held as part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, volume 8559 of lecture notes in computer science. Springer, pp 35–51

[Jav16a] Documentation of `java.util.LinkedList`. http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html. Last access Feb 2016.

[Jav16b] Documentation of `java.util.Map`. http://docs.oracle.com/javase/8/docs/api/java/util/Map.html. Last access Feb 2016.

[JBS11] Jensen JB, Birkedal L, Sestoft P (2011) Modular verification of linked lists with views via separation logic. J Object Technol 10(2)1–20

[JK11] Jacobs S Kuncak V (2011) Towards complete reasoning about axiomatic specifications. In: 12th international conference on verification, model checking, and abstract interpretation, VMCAI 2011, Austin, TX, USA, Jan 23–25, 2011. Proceedings, volume 6538 of lecture notes in computer science. Springer, pp 278–293

[JPS06] Jacobs B, Piessens F, Schulte W (2006) VC generation for functional behavior and non-interference of iterators. In: Proceedings of the 2006 conference on specification and verification of component-based systems, SAVCBS'06. ACM, pp 71–74

[JSP+11] Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F (2011) VeriFast: A powerful, sound, predictable, fast verifier for C and Java. NASA Form Methods, pp 41–55

[Kas06] Kassios IT (2006) Dynamic frames: support for framing, dependencies and sharing without restrictions. In: FM 2006: formal methods, 14th international symposium on formal methods, Hamilton, Canada, Aug 21–27, 2006. Proceedings, pp 268–283

[KEH+09] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2009) seL4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM symposium on operating systems principles 2009, SOSP 2009, Big Sky, Montana, USA, Oct 11–14, 2009. ACM, pp 207–220

[KPS10] Kuncak V, Piskac R, Suter P (2010) Ordered sets in the calculus of data structures. In: Computer science logic, 24th international workshop, CSL 2010, 19th annual conference of the EACSL, Brno, Czech Republic, Aug 23–27, 2010. Proceedings, volume 6247 of lecture notes in computer science. Springer, pp 34–48

[KPSW10] Kuncak V, Piskac R, Suter P, Wies T (2010) Building a calculus of data structures. In: 11th international conference on verification, model checking, and abstract interpretation, VMCAI 2010, Madrid, Spain, Jan 17–19, 2010. Proceedings, volume 5944 of lecture notes in computer science. Springer, pp 26–44

[KRJ09] Kawaguchi M, Rondon PM, Jhala R (2009) Type-based data structure verification. In: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009, pp 304–315

[LBR06] Leavens GT, Baker AL, Ruby C (2006) Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw Eng Notes 31(3):1–38

[Lei95] Leino KRM (1995) Toward reliable modular programs. Ph.D. thesis, Caltech

[Lei10] Leino KRM (2010) Dafny: An automatic program verifier for functional correctness. In: 16th international conference on logic for programming, artificial intelligence, and reasoning, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, revised selected papers, volume 6355 of lecture notes in computer science. Springer, pp 348–370

[Ler09] Leroy X (2009) Formal verification of a realistic compiler. Commun ACM 52(7):107–115

[LL11] Laviron V, Logozzo F (2011) Subpolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities. Softw Tools Technol Transf 13(6):585–601

[LM04] Leino KRM, Müller P (2004) 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. Springer, pp 491–516

[LM06] Leino KRM, Müller P (2006) A verification methodology for model fields. In: 15th European symposium on programming—programming languages and systems, ESOP 2006, Held as part of the joint European conferences on theory and practice of software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings, volume 3924 of lecture notes in computer science. Springer, pp 115–130

[LM09] Leino KRM, Müller P (Sept 2009) Using the Spec# language, methodology, and tools to write bug-free programs. http://www.codeplex.com/Download?ProjectName=specsharp&DownloadId=84056,

[LM10a] Leino KRM, Moskal M (2010) Usable auto-active verification. In: Usable verification workshop. http://fm.csl.sri.com/UV10/

[LM10b] Leino KRM, Moskal M (2010) VACID-0: Verification of ample correctness of invariants of data-structures, 0 edn. VSTTE Workshops, http://goo.gl/0VnvyO

[Loc13] Lochbihler A (2013) Light-weight containers for Isabelle: efficient, extensible, nestable. In: 4th international conference on interactive theorem proving, ITP 2013, Rennes, France, July 22–26, 2013. Proceedings, volume 7998 of lecture notes in computer science. Springer, pp 116–132

[LP13] Leino KRM, Polikarpova N (2013) Verified calculations. In: 5th international conference on verified software: theories, tools, experiments, VSTTE 2013, Menlo Park, CA, USA, May 17–19, 2013, revised selected papers, pp 170–190

[LPZ02] Leino KRM, Poetzsch-Heffter A, Zhou Y (2002) 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, pp 246–257

[LQ08] Lahiri SK, Qadeer S (2008) Back to the future: revisiting precise program verification using SMT solvers. In: Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2008, San Francisco, California, USA, Jan 7–12, 2008. ACM, pp 171–182

[LW08] Leino KRM, Wallenburg A (2008) Class-local object invariants. In: Proceeding of the 1st annual India software engineering conference, ISEC 2008, Hyderabad, India, Feb 19–22, 2008, pp 57–66

[Mey97] Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice Hall, Upper Saddle River

[MPHL06] Müller P, Poetzsch-Heffter A, Leavens GT (2006) Modular invariants for layered object structures. Sci Comput Program 62(3):253–286

[MSBS12] Mehnert H, Sieczkowski F, Birkedal L, Sestoft P (2012) Formalized verification of snapshotable trees: separation and sharing. In: 4th International conference on verified software: theories, tools, experiments, VSTTE 2012, Philadelphia, PA, USA, Jan 28–29, 2012. Proceedings, pp 179–195

[Mül02] Müller P (2002) Modular specification and verification of object-oriented programs, volume 2262 of lecture notes in computer science. Springer

[.NE16a] Documentation of `Systems.Collections.Generic.Dictionary`. https://msdn.microsoft.com/en-us/library/xfhwa508.aspx. Last access Feb 2016

[.NE16b] Documentation of `Systems.Collections.Generic.List.Enumerator`. https://msdn.microsoft.com/en-us/library/x854yt9s.aspx. Last access Feb 2016

[NMS+08] Nanevski A, Morrisett G, Shinnar A, Govereau P, Birkedal L (2008) Ynot: dependent types for imperative programs. In: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, Sept 20–28, 2008. ACM, pp 229–240

[Pau11] Paulin-Mohring C (2011) Introduction to the Coq proof-assistant for practical software verification. In: Tools for practical software verification, LASER, international summer school 2011, Elba Island, Italy, revised tutorial lectures, volume 7682 of lecture notes in computer science. Springer, pp 45–95

[PB08] Parkinson MJ and Bierman GM (2008) Separation logic, abstraction and inheritance. In: Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2008, San Francisco, California, USA, Jan 7–12, 2008. ACM, pp 75–86

[PFM10] Polikarpova N, Furia CA, Meyer B (2010) Specifying reusable components. In: Proceedings of the 3rd international conference on verified software: theories, tools, and experiments (VSTTE'10), volume 6217 of lecture notes in computer science. Springer, pp 127–141

[Pol14] Polikarpova N (2014) Specified and verified reusable components. Ph.D. thesis, ETH Zurich

[Pol15] Polikarpova N (2015) EiffelBase2 (repository of verified code). http://dx.doi.org/10.5281/zenodo.16520

[PQM14] Pek E, Qiu X, Madhusudan P (2014) Natural proofs for data structure manipulation in C using separation logic. In: ACM SIGPLAN conference on programming language design and implementation, PLDI '14, Edinburgh, UK June 09–11, 2014, pp 46

[PTF15] Polikarpova N, Tschannen J, Furia CA (June 2015) A fully verified container library. In: Proceedings of the 20th international symposium on formal methods (FM), volume 9109 of lecture notes in computer science. Springer, pp 414–434

[PTFM14] Polikarpova N, Tschannen J, Furia CA, Meyer B (2014) Flexible invariants through semantic collaboration. In: FM 2014: formal methods—19th international symposium, Singapore, May 12–16, 2014. Proceedings, pp 514–530

[PWZ14] Piskac R, Wies T, Zufferey D (2014) Automating separation logic with trees and data. In: 26th international conference on computer aided verification, CAV 2014, Held as part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, volume 8559 of lecture notes in computer science. Springer, pp 711–728

[RP08] Régis-Gianas Y, Pottier F (2008) A Hoare logic for call-by-value functional programs. In: 9th international conference on mathematics of program construction, MPC 2008, Marseille, France, July 15–18, 2008. Proceedings, volume 5133 of lecture notes in computer science. Springer, pp 305–335

[SRW02] Sagiv S, Reps TW, Wilhelm R (2002) Parametric shape analysis via 3-valued logic. ACM Trans Program Lang Syst 24(3):217–298

[SSK11] Suter P, Steiger R, Kuncak V (2011) Sets with cardinality constraints in satisfiability modulo theories. In: 12th international conference on verification, model checking, and abstract interpretation, VMCAI 2011, Austin, TX, USA, Jan 23–25, 2011. Proceedings, volume 6538 of lecture notes in computer science. Springer, pp 403–418

[TFNP15] Tschannen J, Furia CA, Nordio M, Polikarpova N (2015) AutoProof: Auto-active functional verification of object-oriented programs. In: 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. Springer, pp 566–580

[Ver16] Verifast example gallery. http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/. Last access Feb 2016

[VSJ14] Vazou N, Seidel EL, Jhala R (2014) LiquidHaskell: experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN symposium on haskell, Haskell'14, New York, NY, USA. ACM, pp 39–51

[WEH+96] Weide B, Edwards S, Heym WD, Long T, and Ogden W (April 1996) Characterizing observability and controllability of software components. In: Proceedings fourth international conference on software reuse, 1996, pp 62–71

[Why16] Why3 example gallery. http://toccata.lri.fr/gallery/why3.en.html. Last access Feb 2016.

[WMK11] Wies T, Muñiz M, Kuncak V (2011) An efficient decision procedure for imperative tree data structures. In: Automated deduction—CADE-23—23rd international conference on automated deduction, Wroclaw, Poland, July 31 Aug 5, 2011. Proceedings, volume 6803 of lecture notes in computer science. Springer, pp 476–491

[WMK12] Wies T, Muñiz M, Kuncak V (2012) Deciding functional lists with sublist sets. In: 4th international conference on verified software: theories, tools, experiments, VSTTE 2012, Philadelphia, PA, USA, Jan 28–29, 2012. Proceedings, volume 7152 of lecture notes in computer science. Springer, pp 66–81

[XP99] Xi H, Pfenning F (1999) Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'99, New York, NY, USA. ACM, pp 214–227

[YLB+08] Yang H, Lee O, Berdine J, Calcagno C, Cook B, Distefano D, O'Hearn PW (2008) Scalable shape analysis for systems code. In: 20th international conference Computer Aided Verification, CAV 2008, Princeton, NJ, USA, July 7–14, 2008. Proceedings, volume 5123 of lecture notes in computer science. Springer, pp 385–398

[ZKR08] Zee K, Kuncak V, Rinard MC (2008) Full functional verification of linked data structures. In: Proceedings of the ACM SIGPLAN 2008 conference on programming language design and implementation, Tucson, AZ, USA, June 7–13, 2008, pp 349–361

Received 9 March 2016
Accepted in revised form 25 June 2017 by Frank de Boer, Nikolaj Bjorner, and Andrew Butterfield
Published online 20 September 2017