A Fully Verified Container Library

Nadia Polikarpova[1(✉)], Julian Tschannen[2], and Carlo A. Furia[2]

[1] MIT CSAIL, Cambridge, USA
polikarn@csail.mit.edu
[2] Department of Computer Science, ETH Zurich, Zürich, Switzerland
{Julian.Tschannen,Carlo.Furia}@inf.ethz.ch

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, 8,400 LOC) is possible with moderate annotation overhead (1.4 lines of specification per LOC) and good performance (0.2 seconds per method on average).

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" [63] 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 of a realistic, object-oriented container library.

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 of research on functional verification of heap-manipulating programs and its applications to data structure implementations, to our knowledge, no previous work has 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

Work partially supported by SNF grants 200021-137931 (FullContracts), 200020-134974 (LSAT), and 200021-134976 (ASII); and by ERC grant 291389 (CME).
N. Polikarpova–Work done mainly while affiliated with ETH Zurich.

© Springer International Publishing Switzerland 2015
N. Bjørner and F. de Boer (Eds.): FM 2015, LNCS 9109, pp. 414–434, 2015.
DOI: 10.1007/978-3-319-19249-9_26

examples from recent work in this area (see Sec. 5 for more): Zee et al. [63] 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. [47] 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. [29] 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.

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 Sec. 4, consists of over 8,000 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 2,000 lines of client code that uses some of EiffelBase2's containers.

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 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 [30,40]. Even data structure verification uses interactive provers, such as in [63], 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* [36]; examples are Dafny [35], VCC [12], and VeriFast [24], as well as AutoProof, which we developed in previous work [52,56] 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* [52]: 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 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 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.

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 Sec. 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 [50])

For brevity, the paper focuses on presenting EiffelBase2's verification effort and the new features of AutoProof that we introduced to this end; our previous work [51,52,56] supplies complementary and background technical details.

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

Interface Specifications. Each class in EiffelBase2 declares its abstract state through a set of `model` attributes. As shown in Fig. 1, the model of class `LINKED_LIST` is a `sequence` of list elements. Its type `MML_SEQUENCE` is from the Mathematical Model Library (MML); instances of MML model classes are mathematical values that have custom logical representations in the underlying prover.

Commands—methods with observable side effects, such as `extend_back`—modify the abstract state of objects listed in their frame specification (`modify` clause), according to their postcondition (`ensure` clause). Queries—methods that return a result and have no observable side effect, such as `first`—express, in their postcondition, the return value as a function of the abstract state, which they do not modify. By referring to an explicitly declared model, interface specifications are concise, have a consistent level of

```
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
  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
  seq_refines_bag: bag = sequence.to_bag
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
  target_exists: target ≠ Void
  subjects_definition: subjects = {target}
  index_range: 0 ≤ index ≤ target.sequence.count + 1
  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

abstraction, and can be checked for completeness (whether they uniquely characterize the results of queries and the effect of commands on the model state [51]).

Abstract specifications are convenient for clients, which can reason about the effect of method calls 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.

Connecting Abstract and Concrete State. Verifying the implementation of `first` in Fig. 1 requires relating the model of the list to its concrete representation. We accomplish this through the class **`invariant`**: the clause named `sequence_implementation` asserts that model attribute `sequence` lists the `items` stored in the chain of `LINKABLE` nodes denoted as `cells`; `cells`, in turn, is related to the concrete heap representation by invariant clauses `cells_first` and `cells_linked`.

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. Following the approach introduced with Spec# [37,2], our methodology equips every object with a built-in ghost[1] Boolean attribute `closed`. Whenever an object is closed (`closed` is true), its invariant must hold; but when it is open (`closed` is false), its invariant may not hold. Built-in ghost methods `unwrap` and `wrap` mediate opening and closing objects: `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 a call **`Current`**`.unwrap`[2] at the beginning of every public command; a call **`Current`**`.wrap` at the end of the command; and an assertion **`Current`**`.closed` to the command's pre- and postcondition; defaults can be overridden to implement more complex behavior.

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. In practice, however, the cells are part of the list's internal representation, and should not be directly accessible to other clients. For such *hierarchical* object dependencies, AutoProof implements an *ownership* scheme [37,12]: each object x includes a ghost set `owns` of "owned" objects on which x may depend. AutoProof prevents objects in x`.owns` from being opened (and hence, modified) as long as x is closed; thus, x's consistency cannot be indirectly broken. `LINKED_LIST`'s invariant clause `owns_definition` asserts that the list owns precisely its `cells`, thus allowing the following clauses to depend on the state of the cells.

Safe Iterators. Like other container libraries, 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

[1] Ghost code only belongs to specifications; see Sec. 3.2 for details.

[2] In Eiffel, **`Current`** denotes the receiver object (**`this`** in Java).

invalidate the other iterators. This is not only a challenge to verification but a practical programming problem. To address it, Java's `java.util` iterators implement *fail-safe* behavior, which amounts to checking for validity at every iterator usage, raising an exception whenever the check fails. 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" [26]. In contrast, through complete specifications, EiffelBase2 offers robust *safe* iterators: clients reason precisely about correct usage statically, so that safe behavior will follow without runtime overhead. Fig. 1 shows excerpts from EiffelBase2's linked list iterators.

Collaborative Invariants. Object dependencies such as those arising between a list and its iterators do not quite fit hierarchical ownership schemes: 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 [52]. In AutoProof, each object x is equipped with the ghost sets `subjects` and `observers`: x.`subjects` contains the objects x may depend on (such as an iterator's `target` list); x.`observers` contains the objects that may depend on x. AutoProof verifies that `subjects` and `observers` are consistent between dependent objects (any subject of x has x as an observer), and that any update to a subject does not affect the consistency of its observers. `LINKED_LIST_ITERATOR`'s invariant clause `subjects_definition` asserts that the iterator might depend on its `target` list; correspondingly, the list has to include all active iterators among its `observers`, which is established in the iterator's constructor by calling `target.add_iterator`. The precondition of `LINKED_LIST.extend_back` requires that all the list's `observers` be open: this way, the list can be updated without running the risk of breaking invariants of closed iterators.

2.2 Hash Table

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" [27]. In contrast, EiffelBase2's specification precisely captures which key modifications affect consistency, ensuring safe behavior without restricting usage scenarios. Fig. 2 shows excerpts from EiffelBase2's hash table and key management classes.

Shared Ownership. A table's consistency depends on its keys, but this dependency fits neither ownership nor collaboration: 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*

```
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
  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]
  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
  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`

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 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. Clients can also modify keys as long as the invariant of the keys' lock is maintained. Note that `LOCK` is a **`ghost`** class: its state and operations are absent from the compiled code, and thus incur no runtime overhead.

3 Verification Approach

AutoProof works by translating annotated Eiffel code into the Boogie intermediate verification language [1], and uses the Boogie verifier to generate verification conditions, which are then discharged by the SMT solver Z3. 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 presents the main new (with respect to our previous work [52]) features of both the methodology and the tool.

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* [8]: a purely applicative class whose semantics for verification is given

by a collection of 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 [49, 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. Ghost state has to be updated inside method bodies; the overhead of such updates becomes burdensome in realistic code as ghost variables proliferate, even though their relation to physical program state mostly remains straightforward. 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` before every call to **`Current`**`.wrap`. In Fig. 1, for example, invariant clause `seq_refines_bag` gives rise to the 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 refined version, `sequence`.

3.3 Model-Based Specifications

As illustrated in Sec. 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. Model attributes play a special role in frame specifications: a method annotated with the clause **`modify model`** `s`$[m_1,\ldots,m_n]$ can only modify attributes $m_1,\ldots,m_n$ in the abstract state of `s`, but has no direct restrictions on modifying the concrete state of `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, similar to data groups [38].

Declaring a model also makes it possible to reason about the *completeness* of interface specifications [51]. 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 defines the returned result as a function of the model; the model of a class `C` is complete if it supports complete specifications of all public methods in `C`, such that different abstract states are distinguishable by public method calls. 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; conversely, a sequence is not a complete model for a class `SET` because its interface provides no methods that discriminate element ordering. AutoProof currently does not support mechanized completeness proofs; however, we found that even reasoning informally about completeness—as we did in the design of EiffelBase2—helps provide clear guidelines for writing interface specifications and substantiates the notion of "full functional correctness".

3.4 Inheritance

Postconditions and invariants can be strengthened in descendant classes; hence, any verifier that supports inheritance has to ensure that inherited methods do not violate strengthened invariants, or are appropriately overridden [43,46].

In AutoProof, method implementations can be declared *covariant* or *nonvariant*. A nonvariant implementation cannot depend on the dynamic type of the receiver, 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 knowledge that an object *is* consistent from the *details* of its invariant promotes modular verification. This feature is a boon of invariant-based reasoning; while a similar decoupling is achievable in separation logic through abstract predicates and predicate families [45], it is missing in other approaches such as dynamic frames [28].

3.5 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. We summarize some finicky features of the translation that are crucial for performance.

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 $\mathtt{I}_1,\ldots,\mathtt{I}_n$ are checked; the naive encoding **assert** $\mathtt{I}_1;\ldots;$ **assert** $\mathtt{I}_n$ does not work well for complex invariants: for $j < k$, $\mathtt{I}_k$ normally does not depend on $\mathtt{I}_j$, and hence the previously established fact that $\mathtt{I}_j$ holds just clutters the proof space. Instead, we adopt Dafny's calculational proof approach [39] 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 invari-

ant; `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).

Opaque Functions are pure functions whose axiomatic definitions can be selectively introduced only when needed.[3] 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 complex invariant clauses that are rarely needed in proofs.

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 [51]. Library versions predating our verification effort have been used in introductory programming courses since 2011, 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 outlined in Sec. 2, 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 Sec. 3.5. 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.[4]

Verified EiffelBase2 consists of 46 classes offering an API with 135 public methods; its implementation has over 8,000 lines of code and annotations in 79 abstract and 378 concrete methods. The bulk of the classes belong to one of two hierarchies: containers

[3] A similar concept was independently developed for Dafny at around the same time [11].

[4] 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.

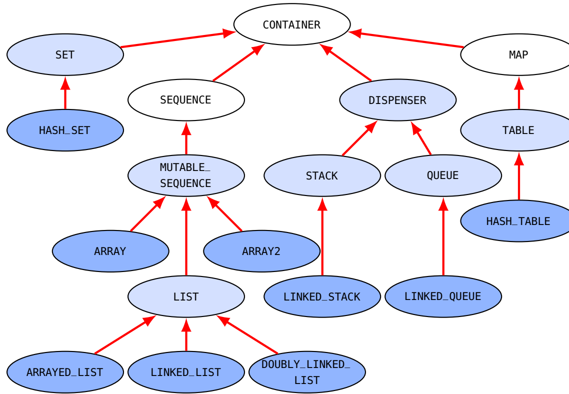

Fig. 3. EiffelBase2 containers: arrows denote inheritance; abstract classes have a lighter background (white for classes with immutable interfaces)

(Fig. 3) and iterators. Extensive usage of inheritance (including multiple inheritance) makes for uniform abstract APIs and reusable implementations.

Completeness. All 135 public methods have complete functional specifications according to the definition given in Sec. 3.3. 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 suitable annotations (described below), AutoProof verifies all 378 method implementations automatically.

Bugs Found. A byproduct of verification was exposing 3 subtle bugs: a division by zero resulting from conversions between machine integers of different sizes; wrong results when moving, in the same array, a range of elements to an overlapping range with a smaller offset; an incorrect implementation of subrange equality in a low-level array service class (wrapping native C arrays) used by EiffelBase2. 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 [49, 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			LOC						TOK	TIME (SEC)			
	ABS	CONC	VER	TOTAL	EXEC	SPEC	REQ	AUX	SPEC/EXEC	SPEC/EXEC	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**

Specification Succinctness. Tab. 1 details the size of EiffelBase2's specifications: overall, 1.4 lines of annotations per line of executable code. The same overhead in tokens—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 [47] verified list implementations have overheads of 0.6 (LOC) and 2.6 (tokens)[5]; 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 [63] 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 [58] has overheads of 4.4 (LOC) and 10.1 (tokens).

Kinds of Specifications. [47] suggests 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 Sec. 3.5) and *structural* annotations (all other auxiliary annotations). Suggestions roughly correspond to "level-C annotations" in [47], in that they are hints to help AutoProof verify more quickly. 12% of all EiffelBase2's specifications 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.

Default Annotations. help curb the annotation overhead. Default wrapping calls (Sec. 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 (Sec. 3.2) always work.

Client Reasoning. To demonstrate that EiffelBase2's interface specifications enable client reasoning, we verified parts of three gaming applications (a transportation system simulator and two board games) that were implemented to support teaching computer

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

science courses and were written before verifying EiffelBase2. The applications total 37 classes and 2,040 lines of code. Their program logics rely on arrays, lists, streams, and tables from EiffelBase2; we focused on verifying correctness of the interactions between library and applications (for example, iterator safety). Annotating the clients required relatively little effort—roughly three person-days to produce around 1,700 lines of annotations—and only trifling modifications to the code. Verification of 84% of over 200 methods succeeded; the exceptions were in large classes that use up to 7 complex data structures simultaneously, where accumulated specification complexity bogs down AutoProof, which times out; verifying these complex parts would require restructuring the code to improve its modularity.

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 value of the 95th percentile.

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 Tab. 1, 47 ghost methods and lemmas, well-formedness of each class invariant, and 56 inherited methods that are covariant (Sec. 3.4) and hence must be re-verified; on the other hand, nonvariant annotations avoid re-verification of 343 bodies, and hence save 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 seconds; over 89% in under 1 second; the most complex method verifies in under 12 seconds. These uniform, short verification times are a direct result of AutoProof's flexible approach to verification, and specifically of our effort to provide an effective Boogie encoding; for example, independent checking of invariant clauses (Sec. 3.5) halves the verification time of some of the most complex methods.

4.3 Challenges

Sec. 2 outlined two challenging features of realistic, general-purpose libraries (***safe iterators*** and ***custom mutable keys***); we now discuss other general challenging aspects.

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 routinely evaded in verification.

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 (e.g., `seq_refines_bag` in Fig. 1).

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 [17]; 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

Well-defined interfaces make verifying *client* code using containers somewhat simpler than verifying container implementations. Techniques used to this end include symbolic execution [20], model checking [5], interactive provers [16], and static analysis [15].

Verification of individual data structures demonstrates that a tool or technique can address fundamental challenges; but also normally abstracts away details that are crucial in realistic general-purpose implementations such as EiffelBase2. Individual data structure challenges have been tackled using several of the major functional verification tools out there, including Why3 [59], Pangolin [53], VeriFast [58], GRASShoper [48], ACL2 [18], Dafny [14], KeY [6,19], Coq [42], and other approaches based on direct constraint solving such as [33].

Data structure collections in functional 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 [22,41] verify the correctness of a high level abstract model, which is then extended into correct-by-construction executable code. The rich type systems of functional languages support mechanisms such as recursive and polymorphic type refinements [57], which naturally capture functional correctness invariant properties of data structures. [29] applied them to verify ML implementations of lists, vectors, maps, and trees. [29]'s techniques are completely automatic and require very little annotations; but they are not directly applicable to data structures that are cyclic and allow arbitrary access patterns, or that are not defined in functional programming style.

Data structure collections in heap-based languages. Different techniques target different trade-offs between automation and expressiveness of specifications to be verified. *Simple properties* can be verified automatically with little or no annotations: absence of

errors such as out of bound array accesses, null dereferences, buffer overruns, and division by zero [34], basic array properties [13], and reachability of objects in the heap (shape analysis) [54,3,4,62,7]. Within the limits of the properties they can express, these analysis techniques are applicable to realistic implementations in real programming languages. Fully automatic techniques have been gradually extended to cover some *decidable functional specification abstractions* such as sets and bags. Some works [21,9,23] are based on top of shape analysis. Others [31,32,60,25,55,61] target logic fragments amenable to SMT reasoning. These decidable abstractions capture essential traits of the interface behavior of data structures, but cannot exactly express the semantics of complex operations with arbitrary element access order.

In contrast, fully *interactive* techniques have no a priori limitations on the properties that can be reasoned upon, but require expert users who can provide low-level proof details. [44,10], for example, reason in higher-order separation logic about sharing and aliasing of data structures featuring a mix of functional and heap-based constructs; such a great flexibility brings a significant overhead in terms of proof scripts.

Auto-active verification [36] tries to provide a high degree of automation, but without sacrificing the expressiveness needed for full functional correctness. Zee et al. [63] document a landmark result in verifying full functional correctness of a significant collection of complex data structures 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 [63] does not always consider general-purpose implementations (for example, hash tables only offer reference-based key comparison, which is too limiting in practice), nor does it target a unitarily designed library. Pek et al.'s [47] natural proofs 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 (Sec. 4.2). They demonstrate their VCDryad tool on complex data structures including singly and doubly linked lists, and trees; some implementations are taken from C's `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 [47] does 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.

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 [11]. 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" [63] 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

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
3. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
4. Beyer, D., Henzinger, T.A., Théoduloz, G., Zufferey, D.: Shape refinement through explicit heap analysis. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 263–277. Springer, Heidelberg (2010)
5. Blanc, N., Groce, A., Kroening, D.: Verifying C++ with STL containers via predicate abstraction. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, USA, November 5-9, pp. 521–524 (2007)
6. Bruns, D.: Specification of red-black trees: Showcasing dynamic frames, model fields and sequences. In: Ahrendt, W., Bubel, R. (eds.) 10th KeY Symposium, Nijmegen, the Netherlands (2011), Extended Abstract
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6), 26 (2011)
8. Charles, J.: Adding native specifications to JML. In: Workshop on Formal Techniques for Java-like Programs, (FTFJP) (2006)

9. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. 77(9), 1006–1036 (2012)
10. Chlipala, A., Gregory Malecha, J., Morrisett, G., Shinnar, A., Wisnesky, R.: 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, August 31- September 2, pp. 79–90. ACM (2009)
11. Christakis, M., Leino, K.R.M., Schulte, W.: Formalizing and verifying a modern build language. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 643–657. Springer, Heidelberg (2014)
12. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
13. Cousot, P., Cousot, R., Logozzo, F.: 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, January 26-28, pp. 105–118. ACM (2011)
14. Dafny example gallery, http://dafny.codeplex.com/SourceControl/latest (last access: November 2014)
15. Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 187–200. ACM, New York (2011)
16. Dross, C., Filliâtre, J.-C., Moy, Y.: Correct code containing containers. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 102–118. Springer, Heidelberg (2011)
17. Filliâtre, J.-C., Paskevich, A., Stump, A.: The 2nd verified software competition: Experience report. In: COMPARE. CEUR Workshop Proceedings, vol. 873, CEUR-WS.org (2012), https://sites.google.com/site/vstte2012/compet
18. Gamboa, R.A.: A formalization of powerlist algebra in ACL2. J. Autom. Reasoning 43(2), 139–172 (2009)
19. Gladisch, C., Tyszberowicz, S.: Specifying a linked data structure in JML for formal verification and runtime checking. In: Iyoda, J., de Moura, L. (eds.) SBMF 2013. LNCS, vol. 8195, pp. 99–114. Springer, Heidelberg (2013)
20. Gregor, D., Schupp STLlint, S.: lifting static checking from languages to libraries. Softw., Pract. Exper. 36(3), 225–254 (2006)
21. Gulwani, S., McCloskey, B., Tiwari, A.: 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, January 7-12, pp. 235–246. ACM (2008)
22. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data representation synthesis. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 38–49. ACM, New York (2011)
23. Itzhaky, S., Bjørner, N., Reps, T., Sagiv, M., Thakur, A.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 35–51. Springer, Heidelberg (2014)
24. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)

25. Jacobs, S., Kuncak, V.: Towards complete reasoning about axiomatic specifications. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 278–293. Springer, Heidelberg (2011)
26. Documentation of `java.util.LinkedList`, http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html (last access: December 2014)
27. Documentation of `java.util.Map`, http://docs.oracle.com/javase/8/docs/api/java/util/Map.html (last access: December 2014)
28. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
29. Kawaguchi, M., Rondon, P.M., Jhala, R.: 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, pp. 304–315 (2009)
30. 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.: seL4: Formal verification of an OS kernel. In: SOSP, pp. 207–220. ACM (2009)
31. Kuncak, V., Piskac, R., Suter, P.: Ordered sets in the calculus of data structures. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 34–48. Springer, Heidelberg (2010)
32. Kuncak, V., Piskac, R., Suter, P., Wies, T.: Building a calculus of data structures. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 26–44. Springer, Heidelberg (2010)
33. Lahiri, S.K., Qadeer, S.: 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, January 7-12, pp. 171–182. ACM (2008)
34. Laviron, V., Logozzo, F.: Subpolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities. STTT 13(6), 585–601 (2011)
35. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
36. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010), http://fm.csl.sri.com/UV10/
37. M. Leino, K.R., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
38. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: 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, pp. 246–257 (2002)
39. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 170–190. Springer, Heidelberg (2014)
40. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
41. Lochbihler, A.: Light-weight containers for Isabelle: Efficient, extensible, nestable. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 116–132. Springer, Heidelberg (2013)
42. Mehnert, H., Sieczkowski, F., Birkedal, L., Sestoft, P.: Formalized verification of snapshotable trees: Separation and sharing. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 179–195. Springer, Heidelberg (2012)
43. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. 62(3), 253–286 (2006)

44. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 20-28, pp. 229–240. ACM (2008)
45. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, pp. 247–258 (2005)
46. Parkinson, M.J., Bierman, G.M.: 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, January 7-12, pp. 75–86. ACM (2008)
47. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, June 09-11, p. 46 (2014)
48. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Heidelberg (2014)
49. Polikarpova, N.: Specified and Verified Reusable Components. PhD thesis, ETH Zurich (2014)
50. Nadia Polikarpova. EiffelBase2 (repository of verified code) (2015), http://dx.doi.org/10.5281/zenodo.16520
51. Polikarpova, N., Furia, C.A., Meyer, B.: Specifying reusable components. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 127–141. Springer, Heidelberg (2010)
52. Polikarpova, N., Tschannen, J., Furia, C.A., Meyer, B.: Flexible invariants through semantic collaboration. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 514–530. Springer, Heidelberg (2014)
53. Régis-Gianas, Y., Pottier, F.: A Hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008)
54. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
55. Suter, P., Steiger, R., Kuncak, V.: Sets with cardinality constraints in satisfiability modulo theories. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 403–418. Springer, Heidelberg (2011)
56. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015)
57. Vazou, N., Seidel, E.L., Jhala, R.: LiquidHaskell: Experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Haskell 2014, pp. 39–51. ACM, New York (2014)
58. Verifast example gallery, http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/ (last access: November 2014)
59. Why3 example gallery, http://toccata.lri.fr/gallery/why3.en.html (last access: November 2014)
60. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476–491. Springer, Heidelberg (2011)

61. Wies, T., Muñiz, M., Kuncak, V.: Deciding functional lists with sublist sets. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 66–81. Springer, Heidelberg (2012)
62. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
63. Zee, K., Kuncak, V., Rinard, M.C.: 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, pp. 349–361 (2008)