Flexible Invariants Through Semantic Collaboration[⋆]

Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer

Department of Computer Science, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Modular reasoning about class invariants is challenging in the presence of collaborating objects that need to maintain global consistency. This paper presents *semantic collaboration*: a novel methodology to specify and reason about class invariants of sequential object-oriented programs, which models dependencies between collaborating objects by semantic means. Combined with a simple ownership mechanism and useful default schemes, semantic collaboration achieves the flexibility necessary to reason about complicated inter-object dependencies but requires limited annotation burden when applied to standard specification patterns. The methodology is implemented in AutoProof, our program verifier for the Eiffel programming language (but it is applicable to any language supporting some form of representation invariants). An evaluation on several challenge problems proposed in the literature demonstrates that it can handle a variety of idiomatic collaboration patterns, and is more widely applicable than the existing invariant methodologies.

1 The Perks and Pitfalls of Invariants

Class invariants[1] are here to stay [21]—even with their tricky semantics in the presence of callbacks and inter-object dependencies, which make reasoning so challenging [16]. The main reason behind their widespread adoption is that they formalize the notion of *consistent* class instance, which is inherent in object-orientated programming, and thus naturally present when reasoning, even informally, about program behavior.

The distinguishing characteristic of invariant-based reasoning is *stability*: it should be impossible for an operation m to violate the invariant of an object o without modifying o itself. Stability promotes information hiding and simplifies client reasoning about preservation of consistency: without invariants a client would need to know which other objects o's consistency depends on, while with invariants it is sufficient that it checks whether m modifies o—a piece of information normally available as part of m's specification. The goal of an *invariant methodology* (also called *protocol*) is thus to achieve stability even in the presence of inter-object dependencies—where the consistency of o depends on the state of other objects, possibly recursively or in a circular fashion (see Sect. 2 for concrete examples).

The numerous methodologies introduced over the last decade, which we review in Sect. 3, successfully relieve several difficulties involved in reasoning with invariants; but

⋆ Work partially supported by SNF grants LSAT/200020-134974, ASII/200021-134976, and FullContracts/200021-137931; and by Hasler-Stiftung grant #2327.

1 Also known under the names "object invariants" or "representation invariants".

there is still room for improvement in terms of flexibility, usability, and automated tool support. In this paper, we present *semantic collaboration* (SC): a novel methodology for specifying and reasoning about invariants in the presence of inter-object dependencies that combines flexibility and usability and is implemented in a program verifier.

A standard approach to inter-object invariants is based on the notion of *ownership*, which has been deployed successfully in several invariant methodologies [2,10,15] and is available in tools such as Spec# [3] and VCC [4]. Under this model, an invariant of an object *o* only depends on the state of the objects explicitly owned by *o*. Ownership is congenial to object-orientation because it supports a strong notion of encapsulation; however, not all inter-object relationships are hierarchical and hence reducible to ownership. Multiple objects may also *collaborate* as equals, mindful of each other's consistency; a prototypical example is the Observer pattern [6] (see Sect. 2).

Semantic collaboration (introduced in Sect. 4) naturally complements ownership to accommodate invariant patterns involving collaborating objects. Most existing methodologies support collaboration through dedicated specification constructs and syntactic restrictions on invariants [10,1,14,20]; such disciplines tend to work only for certain classes of problems. In contrast, SC relies on standard specification constructs—ghost state and invariants—to keep track of inter-object dependencies, and imposes *semantic* conditions on class invariant representations. Its approach builds upon the philosophy of *locally-checked invariants* (LCI) [5]: a low-level verification method based on two-state invariants. LCI has served as a basis for other specialized, user- and automation-friendly methodologies for ownership and shared-memory concurrency. SC can be viewed as an improved specialization of LCI for object collaboration. To further improve usability, SC comprises useful "defaults", which characterize typical specification patterns.

We implemented SC as part of AutoProof, our automated verifier for the Eiffel object-oriented programming language. The implementation provides more concrete evidence of the advantages of SC compared to other methodologies to specify collaborating objects (e.g., [1,11,20,14] all of which currently lack tool support). We present an experimental evaluation of SC and existing invariant protocols in Sect. 5, based on an extended set of examples, including challenge problems from the SAVCBS workshop series [17]. The evaluation demonstrates that SC is the only methodology that supports (*a)* collaboration with unknown classes, while preserving stability, and (*b)* invariants depending on unbounded sets of objects, possibly unreachable in the heap. The collection of problems of Sect. 5—available at [18] together with our solutions—could serve as a benchmark to evaluate invariant methodologies for non-hierarchical object structures. The website [18] also gives access to the extended version of this paper and to a web interface to AutoProof.

2 Motivating Examples: Observers and Iterators

The *Observer* and *Iterator* design patterns are widely used programming idioms [6], where multiple objects depend on one another and need to maintain a global invariant. Their interaction schemes epitomize cases of inter-object dependencies that ownership cannot easily describe; therefore, we use them as illustrative examples throughout the paper, following in the footsteps of much related work [11,16,14].

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

  update (v: INTEGER)
    do
      value := v
      across subscribers as o do o.notify end
    end

  register (o: OBSERVER) -- Internal
    require
      not subscribers.has (o)
    do
      subscribers.add (o)
    end
end
```

```
class OBSERVER
  subject: SUBJECT
  cache: INTEGER

  make (s: SUBJECT) -- Constructor
    do
      subject := s
      s.register (Current)
      cache := s.value
    end

  notify -- Internal
    do
      cache := subject.value
    end
invariant
  cache = subject.value
end
```

Fig. 1: The *Observer pattern*: an observer's **`invariant`** depends on the state of the `SUBJECT`, which reports its state changes to all its `subscribers`. The clients of the subscribers must be able to rely on their `cache` always being consistent, while oblivious of the update/notify mechanisms that preserve invariants.

***Observer* pattern.** Fig. 1 shows the essential parts of an implementation of the Observer design pattern in Eiffel. An arbitrary number of `OBSERVER` objects (called "subscribers") monitor the public state of a single instance of class `SUBJECT`. Each subscriber maintains a copy of the subject's relevant state (integer attribute `value` in Fig. 1) into one of its local variables (attribute `cache` in Fig. 1). The subscribers' copies are cached values that must be consistent with the state of the subject, formalized as the invariant clause `cache = subject.value` of class `OBSERVER`, which depends on another object's state. This dependency is not adequately captured by ownership schemes, since no one subscriber can have exclusive control over the subject.

In the Observer pattern, consistency is maintained by means of explicit collaboration: the subject has a list of `subscribers`, updated whenever a new subscriber registers itself by calling `register (`**`Current`**`)`[2] on the subject. Upon every change to its state (method `update`), the subject takes care of explicitly notifying all registered subscribers (using an **`across`** loop that calls `notify` on every `o` in `subscribers`). This explicit collaboration scheme—called "considerate programming" in [20]—ensures that the subscribers' state remains consistent (i.e., the class invariant holds) between calls to the public methods of the object structure.

A methodology to verify the Observer pattern must ensure invariant stability; namely, that clients of `OBSERVER` can rely on its invariant without knowledge of the register/notify mechanism. Another challenge is dealing with the fact that the number of subscribers attached to the subject is not fixed a priori, and hence we cannot produce explicit syntactic enumerations of the subscribers' `cache` attributes. We must also be able to verify `update` and `notify` without relying on the class invariant as precondition—in fact, those methods are called on inconsistent objects precisely to restore consistency.

In the ***Iterator* pattern**, an arbitrary number of iterator objects traverse a collection of elements. Fig. 2 sketches an implementation where the `COLLECTION` uses an `ARRAY` of

[2] **`Current`** in Eiffel denotes the current object (**`this`** in Java and C#).

```
class COLLECTION [G]
  count: INTEGER
  elements: ARRAY [G] -- Internal

  add (v: G)
    do ... end

  remove_last
    require
      count >0
    do
      count := count − 1
    end
invariant
  0≤ count and count≤ elements.count
end
```

```
class ITERATOR [G]
  target: COLLECTION [G]
  before, after: BOOLEAN

  item: G
    require
      not (before or after)
    do
      Result := target.elements [index]
    end

  index: INTEGER -- Internal
invariant
  0≤ index and index≤ target.count + 1
  before = index <1
  after = index >target.count
end
```

Fig. 2: The *Iterator pattern*: an iterator's invariant depends on the state of the collection it traverses, which is oblivious of the iterators. Verification must prove that clients do not access disabled iterators, without knowing collection's and iterator's internal states.

`elements` as underlying representation. The `ITERATOR`'s main capability is to return the `item` at the current position `index` in the `target` collection[3]. `item`'s precondition (**`require`**) specifies that this is possible only when the iterator points to a valid element of `target`, that is `index` is between 1 and `target.count` (included); otherwise, if `index` is 0 the iterator is `before` the list, and if it equals `target.count` + 1 it is `after` the list. The invariant of class `ITERATOR` defines the public state components `before` and `after` in terms of the internal state component `index`, as well as the acceptable variability range for `index`.

Since the iterator's invariant depends on the state of the target collection, modifying the collection (for example, by calling `remove_last`) may *disable* the iterator (make it inconsistent). This is aligned with the intended usage of iterators, which should be discarded after traversing a collection without changing it. A verification methodology should ensure that clients of `ITERATOR` only access iterators in a consistent state, without knowledge of the iterator's internal state `index` or of its relation to the `target` collection. An additional obstacle to verification comes from the fact that considerate programming would be at odds with the ephemeral nature of iterators compared to observers: collections are normally implemented unaware of the iterators operating on them; a flexible invariant methodology should allow such implementations.

3 Existing Approaches

A crucial issue is deciding *when* (at which program points) class invariants should hold: state-changing operations normally consist of sequences of elementary updates, which individually may break the class invariant temporarily. To deal with this problem, some methodologies restrict the program points where class invariants are expected to hold; others interpret the invariants in a weakened form, which holds vacuously at intermediate steps during updates (and fully at crucial points).

[3] We omit the description of other necessary operations, such as advancing the iterator, since they are irrelevant for our discussion about invariants.

Methodologies based on **visible-state semantics** [12,7] only require invariants to hold when no operation is being executed on their objects, that is in states visible to clients. Without additional mechanisms, visible-state semantics cannot achieve modularity in the presence of callbacks and inter-object dependencies. Existing solutions adopt aliasing control measures [15] to deal with hierarchical object structures. Other solutions [13,14,20], for collaborative invariants, explicitly indicate which objects might be inconsistent at method call boundaries. These two families of solutions—for hierarchical and for collaborative object structures—based on visible-state semantics are not easily combined; this is a practical limitation, since many object-oriented systems consist of an interplay between both types of structures.

Another family of methodologies, collectively known as **Boogie methodologies** after the program verifier where they have originally been implemented, follow the approach of weakening the default semantics of invariants so that they can be evaluated only when appropriate. In a nutshell, all classes include a ghost Boolean attribute `closed`,[4] which denotes whether an object is in a consistent state; an invariant `inv` is then interpreted as the weaker `closed`⇒`inv`, which vacuously holds for open (i.e., not closed) objects. Methods explicitly indicate whether they expect relevant objects to be closed or open; this approach is more conducive to modularity than visible-state semantics (where a method must list *all* possibly inconsistent objects in the entire program).

The original Boogie methodologies, implemented in the Spec# system [3], are mainly based on *syntactic* mechanisms to express ownership relations. For example, following [2], we would annotate attribute `elements` of class `COLLECTION` in Fig. 2 with `rep`, to denote that it belongs to `COLLECTION`'s internal representation; thus, modifying `elements` is only possible if the `COLLECTION` object owning it has been opened—a situation where `closed`⇒`count` ≤ `elements.count` vacuously holds. This solution only supports representations based on bounded sets of objects known a priori and directly accessible through attributes. Follow-up work [10] partially relaxes these restriction introducing a form of quantification predicating over an `owner` ghost attribute (which goes up the ownership hierarchy), and a mechanism to transfer ownership.

In contrast, the VCC verifier [4] implements a Boogie methodology where ownership is encoded on top of LCI's *semantic* approach [5]. Objects include an additional ghost attribute, `owns`, storing the set of all owned objects; ghost code modifies this set explicitly when the owner object is open. In the example of Fig. 2, instead of annotating attribute `elements` with `rep`, we would introduce a first-order formula, such as `owns` = {`elements`}, in the invariant of `COLLECTION` to express that `elements` is part of the representation. The advantage of this approach becomes apparent with linked structures where owned elements are accessible only by following chains of references (e.g., a linked list owns all reachable cells). In fact, semantic approaches to ownership provide the flexibility necessary to specify an unbounded number of owned objects, which may even be not directly attached to the owner, as well as to implement ownership transfers without need for ad hoc mechanisms. They also simplify the rules of reasoning; for example, invariant admissibility becomes a simple proof obligation that all objects whose state is mentioned in the invariant are bound, by the same invariant, to belong to `owns`. These features have contributed to making VCC applicable to real-world systems [9].

[4] We follow VCC's terminology [4] whenever applicable; other works may use different names.

In addition to ownership, some Boogie methodologies also deal with collaborating objects. [10] introduces the notion of *visibility-based* invariants, which requires that a class be aware of the types and invariants of all objects concerned with its state[5]. For example, in Fig. 1 `SUBJECT` must declare its `value` attribute with a modifier `dependent OBSERVER`. Whenever the subject changes its `value`, it has to check that all potentially affected `OBSERVER`s are open. If aware of the `OBSERVER`'s invariant, it can show that the only affected observers are $\{o\colon \texttt{OBSERVER} \mid o.\texttt{subject} = \textbf{Current}\}$. Such indirect representations of the concerned objects complicate discharging the corresponding proof obligations; and relying on knowing the concerned objects' invariants introduces tight coupling between the collaborating classes. To lift these complications, [1] suggests instead to introduce a ghost attribute `deps` storing the set of all concerned objects. It also introduces *update guards*, allowing a concerned object to state conditions under which its invariant is preserved without revealing the invariant itself. Both approaches [10,1] have shortcomings that derive from their reliance on syntactic mechanisms and conditions: collaboration invariants can only depend on a bounded number of objects known a priori and accessible through attributes (called "pivot fields" in [1]); the types of the concerned objects must be known explicitly; and the numerous ad hoc annotations (e.g., `friend` and `keeping`) and operations (e.g., to modify `deps`) make the methodologies harder to present and use. One of the main goals of our methodology (Sect. 4) is to lift these shortcomings by dealing with collaborative invariants by *semantic* rather than syntactic means—similarly to what VCC did to the classic syntactic treatment of ownership.

Somewhat orthogonally to other Boogie-family approaches, the *history invariants* methodology [11] provides for more loose coupling between the collaborating classes, but gives up stability of invariants.

4 Semantic Collaboration

Our novel invariant methodology belongs to the Boogie family; as we illustrated in Sect. 3, this entails that objects can be *open* or *closed*, and class invariants have to hold only for closed objects. On top of semantic mechanisms for ownership, similar to those developed for VCC (see Sect. 3), our methodology also provides a semantic treatment of dependencies among collaborating objects; hence its name *semantic collaboration*. The keywords and constructs specific to SC are `underlined` in the following.

Overview of semantic collaboration. To specify collaboration patterns, we equip every object `o` with ghost fields `subjects` and `observers`. As their names suggest,[6] `o.subjects` stores the set of objects on which `o`'s invariant might depend; and `o.observers` stores the set of objects potentially concerned with `o` (analogous to `deps` in [1]). The methodology achieves modularity by reducing global validity (all closed objects satisfy their invariants) to local checks of two kinds: *(i)* all concerned objects are stored in `observers`; and *(ii)* updates to the attributes of an object `o` maintain the validity of `o` and

[5] We say that an object o is *concerned* with an attribute a of another object s if updating $s.a$ might affect o's invariant.

[6] While the names are inspired by the Observer pattern, they are also applicable to other collaboration patterns, as we demonstrate in Sect. 4.4. The formatting should avoid confusion.

its observers. Check *(i)* becomes an admissibility condition that every declared class invariant must satisfy. Check *(ii)* holds vacuously for for open observers, thus one way to satisfy it is to "notify" all observers of a potentially destructive update by opening them. For more flexibility the methodology also allows subjects to skip "notifying" observers whenever the attribute update satisfies its *guard* (a notion also inspired by [1]). This option is supported by another admissibility condition: an invariant must remain valid after updates to subjects that comply with their update guards.

4.1 Preliminaries and Definitions

A program is a collection of classes. A class is a collection of attributes, methods, and logical functions (side-effect free and terminating).

Built-in attributes. Every class is implicitly equipped with ghost attributes: `closed` (to encode consistency); `owns` and `owner` (to encode the ownership hierarchy); and `subjects` and `observers` (to encode collaboration). We also define the shorthands: `o.open` for $\neg$`o.closed`; `o.free` for `o.owner.open`; and `o.wrapped` for `o.closed` $\wedge$ `o.free`. The *ownership domain* of an object `o` is $\{$`o`$\}$ if `o` is open, and the transitive closure of `o.owns` if `o` is closed. Attributes `closed` and `owner` are only changed indirectly through the implicitly defined ghost methods `wrap` and `unwrap`, whose semantics is defined below.

Specifications. The specification of a *logical function* consists of a *definition* (a side-effect free expression defining the function value) and a **`read`** clause (an expression that denotes the set of objects on which the value of the function may depend). The specification of a *method* consists of a **`require`** clause (a precondition), an **`ensure`** clause (a postcondition), and a **`modify`** clause (an expression that denotes the set of objects that the method may modify). The specification of a *class* includes its invariant `inv`. The specification of an *attribute* `a` consists of an *update* **`guard`** (a Boolean expression over **`Current`** object, new attribute value `y`, and generic observer object `o`—written guard(**`Current`**`.a` := `y`, `o`)).

Expressions. In addition to the standard programming-language expressions, we support a restricted form of quantification through the syntax **`all`** `x` $\in$ `s` : `B(x)` for universal and **`some`** `x` $\in$ `s` : `B(x)` for existential quantification, where `s` is a set expression and `B(x)` is a Boolean expression over `x`. The special expression **`Void`** (analogous to **`null`** in Java and C#) denotes an object that is always allocated and open.

The *read set* reads(e) of a primitive expression e is defined as follows: for an access `x.a` to attribute `a`, reads(`x.a`) $=$ $\{$`x`$\}$; for a call `x.f (y)` to logical function `f`, reads(`x.f (y)`) is given by the `f`'s **`read`** clause. The read set of a compound expression e is the union of the read sets of e's subexpressions.

The current *heap* H in which expressions are evaluated is normally clear from the context and left implicit. Otherwise, e_h denotes the value of expression e in heap h; and $h[$`x.f` $\mapsto e]$ denotes the heap that agrees with h everywhere except possibly about the value of `x.f`, which is e.

Instructions. For the present discussion, we only have to consider method calls `x.m (y)`, as well as *heap update instructions*: **`create`** `x` (allocate an object and attach it to `x`); `x.a := y` (update attribute `a`); and `x.wrap` and `x.unwrap` (opening and closing an object). The *write set* of an instruction is defined analogously to the read set of an expression, except we take the closure under ownership domains for every method's **`modify`** clause.

4.2 Semantic Collaboration: Goals and Proof Obligations

The **goal** of any invariant methodology is to provide *modular* proof obligations to establish *global* validity: the property that every object in the program is *valid* at every program point. Following SC's approach, an object is valid if satisfies its invariant when closed; thus global validity is defined as:

$$\forall o : o.\underline{\texttt{closed}} \Rightarrow o.\texttt{inv} \tag{G1}$$

Additionally, maintaining ownership-based invariants requires strengthening global validity with the property that whenever a parent object p is closed all its owned objects are closed (and their $\underline{\texttt{owner}}$ attributes point back to p):

$$\forall o, p : p.\underline{\texttt{closed}} \wedge o \in p.\underline{\texttt{owns}} \Rightarrow o.\underline{\texttt{closed}} \wedge o.\underline{\texttt{owner}} = p \tag{G2}$$

Proof obligations. The proof obligations specific to SC consist of two types of checks: *(i)* every class invariant is *admissible* according to Def. 1; and *(ii)* every heap update instruction satisfies its precondition. Sect. 4.3 describes how establishing the proof obligations entails global validity, that is subsumes checking (G1) and (G2).

Definition 1 An invariant `inv` is *admissible* iff:

1. `inv` only depends on **`Current`**, its owned objects, and its subjects:
$$\texttt{inv} \quad \Rightarrow \quad \mathsf{reads}(\texttt{inv}) \subseteq \big(\{\textbf{Current}\} \cup \underline{\texttt{owns}} \cup \underline{\texttt{subjects}}\big) \tag{A1}$$
2. All subjects of **`Current`** are aware of it as an observer:
$$\texttt{inv} \quad \Rightarrow \quad \forall s : s \in \underline{\texttt{subjects}} \Rightarrow \textbf{Current} \in s.\underline{\texttt{observers}} \tag{A2}$$
3. `inv` is preserved by any update $\texttt{s.a} := \texttt{y}$ that conforms to its guard:
$$\forall s, a, y : s \in \underline{\texttt{subjects}} \wedge \texttt{inv} \wedge \mathsf{guard}(s.a := y, \textbf{Current}) \Rightarrow \texttt{inv}_{\mathsf{H}[s.a \mapsto y]} \tag{A3}$$
4. (Syntactic check) `inv` does not mention attributes $\underline{\texttt{closed}}$ and $\underline{\texttt{owner}}$, directly or as part of the definitions of the mentioned logical functions.

The specifications of the heap update instructions are given below; the instructions only modify objects and attributes mentioned in the postconditions.

Allocation creates an open object owned by **`Void`** (and thus free), with no observers:

`create` `x`	**`require`**	**`ensure`**
	`True`	$\texttt{x.}\underline{\texttt{open}} \wedge \texttt{x.}\underline{\texttt{owner}} = \textbf{Void} \wedge \texttt{x.}\underline{\texttt{observers}} = \{\}$

Unwrapping opens a wrapped object:

$\texttt{x.}\underline{\texttt{unwrap}}$	**`require`**	**`ensure`**
	$\texttt{x.}\underline{\texttt{wrapped}}$	$\texttt{x.}\underline{\texttt{open}}$

Attribute update operates on an open object and preserves validity of its observers:

$\texttt{x.a} := \texttt{y}$	**`require`**	**`ensure`**
$(\texttt{a} \neq \underline{\texttt{closed}})$	$\texttt{x.}\underline{\texttt{open}}$	$\texttt{x.a} = \texttt{y}$
	$\textbf{all}\ \texttt{o} \in \texttt{x.}\underline{\texttt{observers}} : \texttt{o.}\underline{\texttt{open}} \vee \mathsf{guard}(\texttt{x.a} := \texttt{y}, \texttt{o})$	

Wrapping closes an open object, whose invariant holds, and gives it ownership over all objects in its $\underline{\texttt{owns}}$ set:

$\texttt{x.}\underline{\texttt{wrap}}$	**`require`**	**`ensure`**
	$\texttt{x.}\underline{\texttt{open}} \wedge \texttt{x.inv}$	$\texttt{x.}\underline{\texttt{wrapped}}$
	$\textbf{all}\ \texttt{o} \in \texttt{x.}\underline{\texttt{owns}} : \texttt{o.}\underline{\texttt{wrapped}}$	$\textbf{all}\ \texttt{o} \in \texttt{x.}\underline{\texttt{owns}} : \texttt{o.}\underline{\texttt{owner}} = \texttt{x}$

4.3 Soundness Argument

The soundness argument has to establish that every program that satisfies the proof obligations of SC is always globally valid, that is satisfies (G1) and (G2). We outline a proof of this fact in three parts. See the extended version [18] for the full proofs.

The first part concerns ownership: every methodology that, like SC, imposes a suitable discipline of wrapping and unwrapping to manage ownership domains reduces (G2) to local checks.

Lemma 1. *Consider a methodology M whose proof obligations verify the following:*

a. freshly allocated objects are $\underline{\texttt{open}}$;
b. whenever $\texttt{x}.\underline{\texttt{owner}}$ is updated or $\texttt{x}.\underline{\texttt{closed}}$ is set to $\textbf{False}$, object $\texttt{x}$ is $\underline{\texttt{free}}$;
c. whenever $\texttt{x}.\underline{\texttt{closed}}$ is updated to $\textbf{True}$, every object $\texttt{o}$ in $\texttt{x}.\underline{\texttt{owns}}$ is $\underline{\texttt{closed}}$ and satisfies $\texttt{o}.\underline{\texttt{owner}} = \texttt{x}$;
d. whenever an attribute $\texttt{x.a}$ (with $\texttt{a} \notin \{\underline{\texttt{closed}}, \underline{\texttt{owner}}\}$) is updated, object $\texttt{x}$ is $\underline{\texttt{open}}$.

Then every program that satisfies M's proof obligations also satisfies (G2) *everywhere.*

Proof (sketch). The proof is by induction on the length of program traces. □

The second part applies to any kind of inter-object invariants and assumes a methodology that, like SC, checks that attribute updates preserve validity of all concerned objects; we show that such checks subsume (G1). How a methodology identifies concerned objects is left unspecified as yet.

Lemma 2. *Consider a methodology M whose proof obligations verify the following:*

a. freshly allocated objects are $\underline{\texttt{open}}$;
b. whenever $\texttt{x}.\underline{\texttt{closed}}$ is updated to $\textbf{True}$, $\texttt{x.inv}$ holds;
c. whenever an attribute $\texttt{x.a}$ (with $\texttt{a} \neq \underline{\texttt{closed}}$) is updated to some $\texttt{y}$, every concerned object satisfies $(\texttt{o}.\underline{\texttt{closed}} \wedge \texttt{o.inv}) \Rightarrow \texttt{o.inv}_{\texttt{H}[\texttt{x.a} \mapsto \texttt{y}]}$;
d. class invariants depend neither on attribute $\underline{\texttt{closed}}$ nor on the allocation status of objects.

Then every program that satisfies M's proof obligations also satisfies (G1) *everywhere.*

Proof (sketch). The proof is by induction on the length of program traces, noting that rule *c* explicitly requires that the validity of all concerned objects be preserved. □

The third part of the soundness proof argues that SC satisfies the hypotheses of Lem. 1 and 2, and hence ensures global validity.

Proposition 3. *Every program that satisfies the proof obligations of SC also satisfies* (G2) *and* (G1) *everywhere.*

Proof (sketch). The crucial part is showing that SC satisfies rule *c* of Lem. 2; namely, that an attribute update $\texttt{x.a} := \texttt{y}$ preserves the invariants of all closed concerned object of $\texttt{x}$. To this end, one proves that all such objects must be contained in $\texttt{x}.\underline{\texttt{observers}}$, which follows from the invariant admissibility conditions (A1) and (A2), and (G2). From the precondition of the update rule and the admissibility condition (A3) it follows that the invariants of all closed observers are preserved by the update. □

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

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

  register (o: OBSERVER) -- Internal
    require
      not subscribers.has (o)
      wrapped
      o.open
    modify Current
    do
      unwrap
      subscribers.add (o)
      observers := observers + { o }
      wrap
    ensure
      subscribers.has (o)
      wrapped
    end
invariant
  observers = subscribers.range
  owns = { subscribers } and subjects = { }
end
```

```
class OBSERVER
  subject: SUBJECT
  cache: INTEGER

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

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

Fig. 3: The *Observer pattern* using SC annotations (underlined).

4.4 Examples

We illustrate SC on the two examples of Sect. 2: Fig. 3 and 4 show the Observer and Iterator patterns fully annotated according to the rules of Sect. 4.2. We use the shorthands `wrap_all` (s) and `unwrap_all` (s) to denote calls to `wrap` and `unwrap` on all objects in a set s. As we discuss in Sect. 5, several annotations of Fig. 3 and 4 are subsumed by the defaults mentioned in Sect. 4.5. We postpone to Sect. 4.6 dealing with update guards and the corresponding admissibility condition (A3).

Observer pattern. The `OBSERVER`'s invariant is admissible (Def. 1) because it ensures that `subject` is in `subjects` (A1) and that **`Current`** is in the `subject`'s `observers` (A2). Constructors normally wrap freshly allocated objects after setting up their state. Public method `update` must be called when the whole object structure is wrapped and makes sure that it is wrapped again when the method terminates. This specification style is convenient for public methods, as it allows clients to interact with the class while maintaining objects in a consistent state, without having to explicitly discharge

```
class COLLECTION [G]
  count: INTEGER
  elements: ARRAY [G] -- Internal

  make (capacity: INTEGER) -- Constructor
    require
      open
      capacity ≥ 0
    modify Current
    do
      create elements(1, capacity)
      owns := { elements } ; wrap
    ensure
      count = 0
      observers = { }
    end

  remove_last
    require
      count >0
      wrapped
      all o ∈ observers : o.wrapped
    modify Current, observers
    do
      unwrap ; unwrap_all (observers)
      observers := { }
      count := count − 1
      wrap
    ensure
      wrapped
      observers = { }
      all o ∈ old observers : o.open
    end
invariant
  0≤ count and count≤ elements.count
  owns = { elements } and subjects = { }
end
```

```
class ITERATOR [G]
  target: COLLECTION [G]
  before, after: BOOLEAN
  index: INTEGER -- Internal

  make (t: COLLECTION) -- Constructor
    require
      open and t.wrapped
    modify Current, t
    do
      target := t ; before := True
      t.unwrap
      t.observers := t.observers + { Current }
      t.wrap
      subjects := { t } ; wrap
    ensure
      target = t
      before and not after
      wrapped
    end

  item: G
    require
      not (before or after)
      wrapped and t.wrapped
    do
      Result := target.elements [index]
    end
invariant
  0≤ index and index≤ target.count + 1
  before = index <1
  after = index >target.count
  subjects = { target }
  target.observers.has (Current)
  observers = { } and owns = { }
end
```

Fig. 4: The *Iterator pattern* using SC annotations (underlined).

any condition. Methods such as `register` and `notify`, with restricted visibility, work instead with open objects and restore their invariants so that they can be wrapped upon return. Since `notify` explicitly ensures `inv`, `update` does not need the precise definition of the observer's invariant in order to wrap it (it only needs to know enough to establish the precondition of `notify`). Thus the same style of specification would work if `OBSERVER` were an abstract class and its subclasses maintained different views of subject's `value`.

Let us illustrate the intuitive reason why an instance of `SUBJECT` cannot invalidate any object observing its state. On the one hand, by the attribute update rule, any change to a subject's state (such as assignment to `value` in `update`) must be reconciled with its `observers`. On the other hand, any closed concerned `OBSERVER` object must be contained in its `subject`'s `observers` set: a subject cannot surreptitiously remove anything from this set, since such a change would require an attribute update, and thus, again, would have to be reconciled with all current members of `observers`.

Iterator pattern. The main differences in the annotations of the Iterator pattern occur in the `COLLECTION` class whose non-ghost state is, unlike `SUBJECT` above, unaware of its `observers`. Method `remove_last` has to unwrap its `observers` according to the update rule. However, it has no way of restoring their invariants (in fact, a collection is in general unaware even of the *types* of the iterators operating on it). Therefore, it can only leave them in an inconsistent state and remove them from the observers set. Public methods of ITERATOR, such as item, normally operate on wrapped objects, and hence in general cannot be called after some operations on the collection has disabled its iterators. The only way out of this is if the client of collection and iterators can prove that a certain iterator object i_x was not in the modified collection's observers; this is possible if, for example, the client directly created i_x. The fact that now clients are directly responsible for keeping track of the observers set is germane to the iterator domain: iterators are meant to be used locally by clients.

4.5 Default Annotations

The annotation patterns shown in Sect. 4.4 occur frequently in object-oriented programs. To reduce the annotation burden in those cases, we suggest some default annotations: for example, to any public procedure (a method not returning values) we add implicit pre- and postcondition that **Current**, its subjects, and its observers be wrapped, as well as implicit ghost instructions to unwrap **Current** at the beginning and wrap it at the end. The defaults are only optional suggestions that can be overridden by providing explicit annotations; this ensures that they do not tarnish the flexibility and semantic nature of our methodology. (See the extended version of this paper for more details.)

4.6 Update guards

Update guards are used to distribute the burden of reasoning about attribute updates between subjects and observers, depending on the intended collaboration scheme. At one extreme, if a $\text{guard}(\texttt{x.a} := \texttt{y}, \texttt{o})$ is identically **False**, the burden is entirely on the subject, which must check that all observers are open whenever a is updated; in contrast, the admissibility condition (A3) holds vacuously for the observer o. At the other extreme, if a guard is identically **True**, the burden is entirely on the observer, which deals with (A3) as a proof obligation that its invariant does not depend on a; in contrast, the subject x can update a without particular constraints.

Another recurring choice for a guard is $\texttt{inv(o)} \Rightarrow \texttt{inv(o)}_{\texttt{H[x.a}\mapsto\texttt{y]}}$. For its flexibility, we chose this as the default guard of SC. Just like **False**, this guard also does not burden the observer, but is more flexible at the other end: upon updating, the subject can establish that each observer is either open or its invariant is preserved. The subject can rely on the latter condition if the observer's invariants are known, and ignore it otherwise.

When it comes to built-in ghost attributes, owns and subjects are guarded with **True**, since other objects are not supposed to depend on them, while observers has a more interesting guard, namely $\text{guard}(\texttt{x.observers} := \texttt{y}, \texttt{o}) = \texttt{o} \in \texttt{y}$. This guard reflects the way this attribute is commonly used in collaboration invariants, while leaving the subject with reasonable freedom to manipulate it; for example, adding new observers to the set observers without "notifying" the existing ones (this is used, in particular, in the register method of Fig. 3).

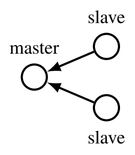
5 Experimental Evaluation

We arranged a collection of representative challenge problems involving inter-object collaboration, and we specified and verified them using our SC methodology. This section presents the challenge problems (Sect. 5.1), and discusses their solutions using SC (Sect. 5.2), as well as other methodologies described in Sect. 3 (Sect. 5.3). See [18] for full versions of problem descriptions, together with our solutions, and a web interface to the AutoProof verifier.

5.1 Challenge Problems

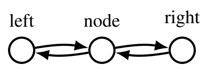
Beside using it directly to evaluate SC, the collection of challenge problems described in this section can be a benchmark for other invariant methodologies. The benchmark consists of six examples of varying degree of difficulty, which capture the essence of various collaboration patterns often found in object-oriented software. The emphasis is on non-hierarchical structures that maintain a global invariant.

We briefly present the six problems in roughly increasing order of difficulty in terms of the shape of references in the heap, state update patterns, and challenges posed to preserving encapsulation. The first two problems in our set are **Observer** [11,16,14,17] and **Iterator** [11,17], which have already been described in Sect. 2.

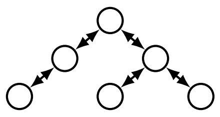


Master clock [1,11]. The time stored by a master clock can increase (public method `tick`) or be set to zero (public method `reset`). The time stored locally by each slave clock must never exceed the master's but need not be perfectly synchronized. Therefore, when the master is `reset` its slaves are disabled until they synchronize (similar to iterators); when the master increments the time its slaves remain in a consistent state without requiring synchronization. *Additional challenges*: `tick`'s frame does not include slaves; perform reasoning local to the master with only partial knowledge of the slaves' invariants.

Variants: a simplified version without `reset` (slaves cannot become inconsistent).



Doubly-linked list [10,13]. The specification expresses the consistency of the `left` and `right` neighbors directly attached to each `node`. Verification establishes that updates local to a node (such as inserting or removing a node next to it) preserve consistency. Unlike in the previous examples, the heap structure is recursive; the main challenge is thus avoiding considering the list as a whole (such as to propagate the effects of local changes).



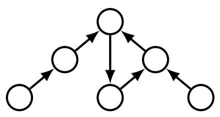
Composite [21,20,8], (see also SAVCBS '08 [17]). A tree structure maintains consistency between the values stored by parent and children nodes (for example, the value of every node is the maximum of its children's). Clients can add children anywhere in the tree; therefore, ownership is unsuitable to model this example. Two new challenges are that the node invariant depends on an unbounded number of children; and that the effects of updates local to a node (such as adding a child) may propagate up the whole tree involving an unbounded

Table 5: The challenge problems specified and verified using SC.

PROBLEM	SIZE (LOC)	TOKENS (no defaults) CODE	REQ	AUX	SPEC/CODE	TOKENS (with defaults) AUX	SPEC/CODE	TIME (sec.)
Observer	129	156	52	296	2.2	185	1.5	8
Iterator	177	168	176	315	2.9	247	2.5	12
Master clock	130	85	69	267	4.0	190	3.1	6
DLL	147	136	83	435	3.8	320	3.0	18
Composite	188	124	270	543	6.6	427	5.6	18
PIP	152	116	310	445	6.5	402	6.1	18
Total	923	785	960	2301	4.2	1771	3.5	80

number of nodes. Specification deals with these unbounded-size footprints; and verification must also ensure that the propagation to restore global consistency terminates. Clients of a tree can rely on a globally consistent state while ignoring the tree structure.

Variations: a simplified version with n-ary trees for fixed n (the number of children is bounded); more complex versions where one can also remove nodes or add subtrees.



PIP [21,20]. The Priority Inheritance Protocol [19] describes a compound whose nodes are more loosely related than in the Composite pattern: each node has a reference to at most one parent node, and cycles are possible. Unlike in the Composite pattern, the invariant of a node depends on the state of objects not directly accessible in the heap (parents do not have references to their children). New challenges derive from the possible presence of cycles, and the need to add children that might already be connected to whole graphs; specifying footprints and reasoning about termination are trickier.

5.2 Results and Discussion

We specified the six challenge problems using SC, and verified the annotated Eiffel programs with AutoProof. Tab. 5 shows various metrics about our solutions: the SIZE of each annotated program; the number of TOKENS of executable CODE, REQuirements specification (the given functional specification to be verified), and AUXiliary annotations (specific to our methodology, both with and without default annotations); the SPEC/CODE overhead, i.e., $(\text{REQ} + \text{AUX})/\text{CODE}$; and the verification time in AutoProof. The overhead is roughly between 1.5 (for Observer) and 6 (for PIP), which is comparable with that of other verification methodologies applied to similar problems. The default annotations of Sect. 4.5 reduce the overhead by a factor of 1.3 on average.

The PIP example is perfectly possible using ghost code, contrary to what is claimed elsewhere [21]. In our solution, every node includes a ghost set `children` with all the child nodes (inaccessible in the non-ghost heap); it is defined by the invariant clause `parent` $\neq$ **`Void`**$\Rightarrow$`parent.children.has (`**`Current`**`)`, which ensures that `children` contains every closed node n such that n`.parent` $=$ **`Current`**. Based on this, the fundamental consistency property is that the `value` of each node is the maximum of the values of nodes in `children` (or a default value for nodes without children), assuming maximum is the required relation between parents and children.

Table 6: Comparison of invariant protocols on the challenge problems.

	VISIBLE-STATE SEMANTICS		BOOGIE METHODOLOGIES			
	Cooperation [14]	Considerate [20]	Spec# [10]	Friends [1]	History [11]	SC
Observer	$\oplus$	$+$	$+$	$\oplus$	$\oplus^{d}$	$\oplus$
Iterator	$-^{a}$	$-^{a}$	$+$	$+$	$\oplus^{d}$	$\oplus$
Master clock	$-^{a}$	$-^{a}$	$+$	$\oplus$	$\oplus^{d}$	$\oplus$
DLL	$+$	$+$	$\oplus$	$+$	$+^{d}$	$\oplus$
Composite	$-^{b}$	$\oplus^{c}$	$-^{b}$	$-^{b}$	$-^{b}$	$\oplus$
PIP	$-^{b}$	$\oplus^{c}$	$-^{b}$	$-^{b}$	$-^{b}$	$\oplus$

^{a} Only considerate programming ^{b} Only bounded set of reachable subjects
^{c} No framing specification ^{d} No invariant stability

The main challenge in Composite and PIP is reasoning about framing and termination of the state updates that propagate along the graph structure. For framing specifications, we use a ghost set `ancestors` with all the nodes reachable following `parent` references. Proving termination in PIP requires keeping track of all visited nodes and showing that the set of ancestors that haven't yet been visited is strictly shrinking.

5.3 Comparison with Existing Approaches

We outline a comparison with existing invariant protocols (discussed in Sect. 3) on our six challenge problems. Tab. 6 reports how each methodology fares on each challenge problem: $-$ for "methodology not applicable", $+$ for "applicable", and $\oplus$ for "applicable and used to demonstrate the methodology when introduced".

Only SC is applicable to all the challenges, and other methodologies often have other limitations (notes in Tab. 6). Most approaches cannot deal with unbounded sets of subjects, and hence are inapplicable to Composite and PIP. The methodology of [20] is an exception as it allows set comprehensions in invariants; however, it lacks an implementation and does not discuss framing, which constitutes a major challenge in Composite and PIP. Both methodologies [14,20] based on visible-state semantics are inapplicable to implementations which do not follow considerate programming; they also lack support for hierarchical object dependencies, and thus cannot verify implementations that rely on library data structures (e.g., Fig. 1 and 2).

Another important point of comparison is the level of coupling between collaborating classes, which we can illustrate using the Master clock example. In [10], class `MASTER` requires complete knowledge of the invariant of class `CLOCK`, which breaks information hiding (in particular, `MASTER` has to be re-verified when the invariant of `CLOCK` changes). The update guards of [1] can be used to declare that slaves need not be notified as long their master's time is increased; this provides abstraction over the slave clock's invariant, but class `MASTER` still depends on class `CLOCK`—where the update guard is defined. In general, the syntactic rules of [1] require that subject classes declare all potential observer classes as "friends". In SC, update guards are defined in subject classes; thus we can prove that `tick` maintains the invariants of all observers without knowing their type. Among the other approaches, only history invariants [11] support the same level of decoupling, but they cannot preserve stability with the `reset` method.

6 Future Work

In an ongoing effort, we have been using SC to verify a realistic data structure library. This poses new challenges to the verification methodology; in particular dealing with *inheritance*. Rather than imposing severe restrictions on how invariants can be strengthened in subclasses, we prefer to re-verify most inherited methods to make sure they still properly re-establish the invariant before wrapping the `Current` object. We maintain that this approach achieves a reasonable trade-off.

References

1. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: MPC. pp. 54–84 (2004)
2. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3 (2004)
3. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM 54(6), 81–91 (2011)
4. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: TPHOLs. LNCS, vol. 5674, pp. 23–42. Springer (2009)
5. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: CAV. pp. 480–494 (2010)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)
7. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, pp. 175–188 (1999)
8. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Asp. Comput. 19(2), 159–189 (2007)
9. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: FM. LNCS, vol. 5850, pp. 806–809 (2009)
10. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: ECOOP. pp. 491–516 (2004)
11. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: ESOP. pp. 80–94 (2007)
12. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 2nd edn. (1997)
13. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Cooperation-based invariants for OO languages. Electr. Notes Theor. Comput. Sci. 160, 225–237 (2006)
14. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Invariants for non-hierarchical object structures. Electr. Notes Theor. Comput. Sci. 195, 211–229 (2008)
15. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. 62(3), 253–286 (2006)
16. Parkinson, M.J.: Class invariants: the end of the road? In: IWACO. ACM (2007)
17. SAVCBS workshop series. `http://www.eecs.ucf.edu/~leavens/SAVCBS/` (2001–2010)
18. Semantic Collaboration website. `http://se.inf.ethz.ch/people/polikarpova/sc/`
19. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput. 39(9), 1175–1185 (1990)
20. Summers, A.J., Drossopoulou, S.: Considerate reasoning and the composite design pattern. In: VMCAI. pp. 328–344 (2010)
21. Summers, A.J., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: IWACO. pp. 1–9. ACM (2009)