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\textsuperscript{1} are here to stay—even with their tricky semantics in the presence of callbacks and inter-object dependencies, which make reasoning so challenging\textsuperscript{1}. 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.\textsuperscript{3} for concrete examples).

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

\textsuperscript{1} Also known under the names “object invariants” or “representation invariants”.

\textsuperscript{*} Work partially supported by SNF grants LSAT/200020-134974, ASII/200021-134976, and FullContracts/200021-137931; and by Hasler-Stiftung grant #2327.
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 [2,11,16] 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 naturally complements ownership to accommodate invariant patterns involving collaborating objects. Most existing methodologies support collaboration through dedicated specification constructs and syntactic restrictions on invariants [11,13,22]; 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. As we argue in Sect. 5 based on several challenge problems, the defaults significantly reduce the annotation burden without sacrificing flexibility in the general case.

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., [1,12,22,15] all of which currently lack tool support).

Outline and contributions. The presentation is based on examples of non-hierarchical object structures, customarily used in the literature. Sect. 2 presents the examples and the challenges they embody; and Sect. 3 discusses the approaches taken by main existing invariant methodologies. Sect. 4 introduces SC, demonstrates its application to the running examples, and outlines a soundness proof. Sect. 5 evaluates both SC and existing protocols on an extended set of examples, including challenge problems from the SAVCBS workshop series [19]. 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 [20] together with our solutions—could serve as a benchmark to evaluate invariant methodologies for non-hierarchical object structures. The website [20] also gives access to AutoProof through a web interface.
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 [12][17][15].

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 regis-
ters itself by calling register (Current) 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 [22]—ensures that the subscribers’ state remains consistent (i.e., the class invariant holds) between calls to the public methods of the object structure.

Fig. 1 uses Eiffel’s selective exports to separate the public interface of the classes from the methods internal to the object structure: feature {OBSERVER} denotes that method register is only available to instances of class OBSERVER, and feature {SUBJECT} similarly limits the visibility of notify to the subject. While selective exports help emphasize collaboration patterns, they are not necessary for the discussion of the present paper, whose results are applicable to any object-oriented language regardless of the available visibility specifiers.

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 elements as underlying representation. The ITERATOR’s main capability is to return the item at the current position index in the target collection. 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. In fact, the selective exports used in Fig. 2 hide the details of ITERATOR’s invariant from its clients (the visibility of an invariant clause is determined by its least visible subexpression, and feature {NONE} denotes purely private members). 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 im-

---

2 Current in Eiffel denotes the current object (this in Java and C#).
3 Similar to friend classes in C++.
4 We omit the description of other necessary operations, such as advancing the iterator, since they are irrelevant for our discussion about invariants.
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. Implemented unaware of the iterators operating on them; a flexible invariant methodology should allow such implementations.

3 Existing Approaches

This section reviews the main existing methodologies for specifying and reasoning about class invariants; based on their most important features and limitations. Sect. 4 will present our own methodology. For lack of space, we only discuss methodologies for inter-object dependencies that support modular reasoning (where local checks on individual classes or small groups of classes subsume global program correctness).

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).

Methodologies based on visible-state semantics only require invariants to hold when no operation is being executed on their objects, that is in states visible to clients. This idea was introduced for Eiffel [13], and later also adopted by JML [8]. Without additional mechanisms, visible-state semantics can’t achieve modularity in the presence of callbacks (the client making the callback is unaware of ongoing operations that
may affect the invariant) and of inter-object dependencies (if \( o_1 \)'s invariant depends on \( o_2 \), the former is also affected by operations on \( o_2 \) invisible to clients of \( o_1 \)). Existing solutions adopt aliasing control measures \([16]\) to deal with hierarchical object structures described by ownership. Other solutions \([14, 15, 22]\), for collaborative invariants, explicitly indicate which objects might be inconsistent at method call boundaries; for example, method \( \text{register}(o : \text{OBSERVER}) \) of class \( \text{SUBJECT} \) in Fig. 1 would be annotated with \( \text{broken} \) to specify that argument \( o \)'s invariant may not hold when executing \( \text{register} \). 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 structure. For example, continuing with Fig. 1, objects of class \( \text{SUBJECT} \) collaborate with \( \text{OBSERVER} \) objects but also own a subscribers list as part of their representation. Thus, when reasoning about method \( \text{register} \), we should be able to deal with the call \( \text{subscribers}\cdot\text{add}(o) \) whose argument \( o \) is inconsistent (and hence \( \text{add} \) cannot assume \( o \)'s invariant); however, annotating \( \text{LIST} \)'s \( \text{add} \) by declaring its argument \( \text{broken} \) goes against modularity, as class \( \text{LIST} \) should not need to know how and where it is used. The difficulty of integrating hierarchical and collaborative models is the main limitation of visible-state methodologies, and likely a reason why, to our knowledge, they have not been implemented in any program verifier.

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 \( \text{closed} \) which denotes whether an object is in a consistent state; an invariant \( \text{inv} \) is then interpreted as the weaker \( \text{closed} \Rightarrow \text{inv} \), 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: it does not impose consistency by default at method call boundaries and thus does not require methods to 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 \( \text{elements} \) of class \( \text{COLLECTION} \) in Fig. 2 with \( \text{rep} \) to denote that it belongs to \( \text{COLLECTION} \)'s internal representation; thus, modifying \( \text{elements} \) is only possible if the \( \text{COLLECTION} \) object owning it has been opened—a situation where \( \text{closed} \Rightarrow \text{count} \leq \text{elements} \cdot \text{count} \) vacuously holds. This solution only supports representations based on bounded sets of objects known a priori and directly accessible through attributes. Follow-up work \([11]\) partially relaxes these restriction introducing a form of quantification predicated over an \( \text{owner} \) ghost attribute (which goes up the ownership hierarchy), and a mechanism to transfer ownership. The additional expressiveness comes with a price to pay mainly in terms of complex invariant admissibility conditions (hence, it may be hard to understand what is expressible and how) and complicated soundness proofs of the methodology.

In contrast, the VCC verifier \([4]\) implements a Boogie methodology where ownership is encoded on top of LCT's semantic approach \([5]\). Objects include an additional

---

5 We follow VCC’s terminology \([4]\) whenever applicable; other works may use different names.
ghost attribute, \texttt{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 \texttt{elements} with \texttt{rep}, we would introduce a first-order formula, such as \texttt{owns} = \{\texttt{elements}\}, in the invariant of \texttt{COLLECTION} to express that \texttt{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 \texttt{owns}. These features have contributed to making VCC applicable to real-world systems [10].

In addition to ownership, some Boogie methodologies also deal with collaborating objects. [11] introduces the notion of \textit{visibility-based} invariants, which requires that a class be aware of the types and invariants of all objects concerned with its state. For example, in Fig. 1, \texttt{SUBJECT} must declare its \texttt{value} attribute with a modifier \texttt{dependent OBSERVER}. Whenever the subject changes its \texttt{value}, it has to check that all potentially affected \texttt{OBSERVERs} are open. If aware of the \texttt{OBSERVER’s} invariant, it can show that the only affected observers are \{\texttt{o: OBSERVER} \mid \texttt{o.subject} = \texttt{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, [11] suggests instead to introduce a ghost attribute \texttt{deps} storing the set of all concerned objects. It also introduces \textit{update guards}, allowing a concerned object to state conditions under which its invariant is preserved without revealing the invariant itself. Both approaches \cite{11,1} 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 [11]); the types of the concerned objects must be known explicitly; and the numerous ad hoc annotations (e.g., \texttt{friend} and \texttt{keeping}) and operations (e.g., to modify \texttt{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 \textit{semantic} rather than syntactic means—similarly to what VCC did to the classic syntactic treatment of ownership. The semantic approach makes SC very flexible, capable of accommodating disparate object-oriented design patterns without requiring ad hoc mechanisms.

Somewhat orthogonally to other Boogie-family approaches, the \textit{history invariants} methodology \cite{12} provides for more loose coupling between the collaborating classes, but gives up stability of invariants.

---

6 We say that an object \(o\) is \textit{concerned} with an attribute \(a\) of another object \(s\) if updating \(s.a\) might affect \(o\)'s invariant.
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, \( o \).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 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

As it is customary, the following presentation targets fundamental constructs, while ignoring those that do not affect reasoning about invariants (e.g., control structures). We also largely ignore issues related to inheritance, but we briefly come back to them in Sect. 6.

A program is a collection of classes. A class is a collection of attributes, methods, and logical functions (side-effect free and terminating). Any of those constructs can be declared ghost if it is meant to be used only in specifications.

**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 \(~o\).closed; `o.free` for `o.owner.open`; and `o.wrapped` for `o.closed \&\& 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 \( \text{inv} \). The specification of an attribute \( \text{a} \) consists of an update guard \( \text{guard} \) (a Boolean expression over \( \text{Current} \) object, new attribute value \( y \), and generic observer object \( o \)—written \( \text{guard}(\text{Current}.a := y, o) \)).

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

The read set \( \text{reads}(e) \) of a primitive expression \( e \) is defined as follows: for an access \( x.a \) to attribute \( a \), \( \text{reads}(x.a) = \{ x \} \); for a call \( x.f(y) \) to logical function \( f \), \( \text{reads}(x.f(y)) \) is given by the \( r \)'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 \( e \). Since we ignore deallocation, our heaps have no dangling references: only allocated objects are reachable from allocated objects.

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 \( x\.\text{wrap} \) and \( x\.\text{unwrap} \) (opening and closing an object).

The write set \( \text{writes}(s) \) of an primitive instruction \( s \) is defined as follows: for an update \( x.a := y \) of attribute \( a \), \( \text{writes}(x.a := y) = \{ x \} \); for opening or closing an object \( x \), \( \text{writes}(x.\text{unwrap}) = \text{writes}(x.\text{wrap}) = \{ x \} \cup x.\text{owns} \); for a call \( x.r(y) \) to method (or constructor) \( r \), \( \text{writes}(x.r(y)) \) is the union of the ownership domains of all objects mentioned in \( r \)'s modify clause. The write set of a compound instruction \( s \) is the union of the write sets of the instructions in \( s \).

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:

\[ \forall o : o.\text{closed} \implies o.\text{inv} \]

(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 \text{owner} attributes point back to \( p \)):

\[ \forall o, p : p.\text{closed} \land o \in p.\text{owns} \implies o.\text{closed} \land o.\text{owner} = p \]

(G2)

Proof obligations. The proof obligations specific to SC consist of two types of checks: (i) every class invariant is admissible according to Def. 4, and (ii) every heap
update instruction satisfies its precondition. These proof obligations are modular in that they only mention the state of the current object, its observers and owned objects. Sect. 4.3 describes how establishing the local proof obligations entails global validity, that is subsumes checking \((G1)\) and \((G2)\).

Admissibility captures the requirements that class invariants respect ownership and collaboration relations, modeled through ghost attributes \(\text{owns}, \text{subjects}, \text{and observers}\).

**Definition 1** An invariant \(\text{inv}\) is admissible iff:

1. \(\text{inv}\) only depends on \(\text{Current}\), its owned objects, and its subjects:
   \[
   \text{inv} \Rightarrow \text{reads}(\text{inv}) \subseteq (\{\text{Current}\} \cup \text{owns} \cup \text{subjects}) \quad (A1)
   \]
2. All subjects of \(\text{Current}\) are aware of it as an observer:
   \[
   \text{inv} \Rightarrow \forall s : s \in \text{subjects} \Rightarrow \text{Current} \in s.\text{observers} \quad (A2)
   \]
3. \(\text{inv}\) is preserved by any update \(s.a := y\) that conforms to its guard:
   \[
   \forall s, a, y : s \in \text{subjects} \wedge \text{inv} \wedge \text{guard}(s.a := y, \text{Current}) \Rightarrow \text{inv}_{H[s.a := y]} \quad (A3)
   \]
4. (Syntactic check) \(\text{inv}\) does not mention attributes \(\text{closed}\) and \(\text{owner}\), 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 \(\text{Void}\) (and thus free), with no observers:

\[
\text{create } x \quad \textbf{require} \quad \text{ensure} \quad \begin{array}{l}
\text{x.open} \wedge x.\text{owner} = \text{Void} \wedge x.\text{observers} = \{
\end{array}
\]

**Unwrapping** opens a wrapped object:

\[
x.\text{unwrap} \quad \textbf{require} \quad \text{ensure} \quad x.\text{wrapped} \wedge x.\text{open}
\]

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

\[
x.a := y \quad \textbf{require} \quad \text{ensure} \quad (a \neq \text{closed}) \quad \begin{array}{l}
x.\text{open} \quad x.a = y \quad \forall o \in x.\text{observers} : o.\text{open} \vee \text{guard}(x.a := y, o)
\end{array}
\]

**Wrapping** closes an open object, whose invariant holds, and gives it ownership over all objects in its \(\text{owns}\) set:

\[
x.\text{wrap} \quad \textbf{require} \quad \text{ensure} \quad x.\text{open} \wedge x.\text{inv} \quad x.\text{wrapped} \quad \begin{array}{l}
\forall o \in x.\text{owns} : o.\text{wrapped} \quad \forall o \in x.\text{owns} : o.\text{owner} = x
\end{array}
\]

**Other proof obligations.** The other proof obligations, which do not involve invariants, are the usual ones of axiomatic reasoning: every call to a method \(m\) occurs in a state that satisfies \(m\)'s precondition; executing a method \(m\) in a state that satisfies its precondition leads to a state that satisfies \(m\)'s postcondition; the \text{read} clause of every logical function \(f\) is consistent (i.e., the read set of \(f\)'s definition is a subset of \(f\)'s \text{read} clause); the \text{modify} clause of every method \(m\) is consistent (i.e., the write set of \(m\)'s body is a subset of \(m\)'s \text{modify} clause); and the definitions of logical functions are terminating.
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.

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 open;

b. whenever \( x.\text{owner} \) is updated or \( x.\text{closed} \) is set to false, object \( x \) is free;

c. whenever \( x.\text{closed} \) is updated to true, every object \( o \) in \( x.\text{owns} \) is closed and satisfies \( o.\text{owner} = x \);

d. whenever an attribute \( x.a \) (with \( a \notin \{\text{closed}, \text{owner}\} \)) is updated, object \( x \) is open.

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

**Proof.** The proof is by induction on the length of program traces.

The base case is the trace only consisting of the initial heap where no object is allocated but for an open object \( \text{Void} \); thus (G2) holds initially. For the inductive step, let \( h \) be the final heap of a trace where (G2) invariably holds. Consider an instruction \( s \) that yields heap \( h' \) if executed on \( h \). Without loss of generality, let \( h' \neq h \); therefore, \( s \) is either an allocation of a new object or an attribute update. If \( s \) allocates a new object \( x \), (G2) still holds in \( h' \): \( x \) is open (rule a) and is in no other object’s owns set, since \( x \) has just been created. If \( s \) is an attribute update, it can only invalidate (G2) if it updates closed, owns, or owner. If \( s \) updates some \( o.\text{owner} \) in (G2)'s consequent or sets \( o.\text{closed} \) to false, then \( o \) is free (rule b); thus \( o.\text{owner} \) is open, and hence (G2)'s antecedent is false. If \( s \) sets to true some \( p.\text{closed} \) in (G2)'s antecedent, then rule c implies the whole (G2) holds. If \( s \) updates some \( p.\text{owns} \) in (G2)'s antecedent, then \( p \) is open (rule d); thus, (G2)'s antecedent is false. \( \square \)

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 open;

b. whenever \( x.\text{closed} \) is updated to true, \( x.\text{inv} \) holds;

c. whenever an attribute \( x.a \) (with \( a \neq \text{closed} \)) is updated to some \( y \), every concerned object satisfies \( (o.\text{closed} \land o.\text{inv}) \Rightarrow o.\text{inv}_{H[x,a\rightarrow y]} \);

d. class invariants depend neither on attribute \( \text{closed} \) nor on the allocation status of objects.

Then every program that satisfies \( M \)'s proof obligations also satisfies (G1) everywhere.
Proof. The proof is by induction on the length of program traces.

The base case is the trace only consisting of the initial heap where no object is allocated but for an open object Void; thus (G1) holds initially. For the inductive step, let \( h \) be the final heap of a trace where (G1) invariably holds. Consider an instruction \( s \) that yields heap \( h' \) if executed on \( h \). Without loss of generality, let \( h' \neq h \); therefore, \( s \) is either an allocation of a new object or an attribute update. If \( s \) allocates a new object \( x \), (G1) still holds in \( h' \): \( x \) is open (rule 3), and no other object’s invariants depends on it, since \( x \) has just been created and class invariant do not know about allocation status (rule 2). If \( s \) sets \( \text{False} \) some \( o.\text{closed} \) in (G1)’s antecedent, then (G1) vacuously hold. If \( s \) sets \( \text{True} \) some \( o.\text{closed} \) in (G1)’s antecedent, then \( o.\text{inv} \) holds (rule 4); thus (G1) holds too. Also, updates to some \( o.\text{closed} \) cannot concern the invariants of objects other than \( o \) (rule 5). If \( s \) updates some \( x.a \), with \( a \neq \text{closed} \), let \( o \) be any object concerned with the update; either \( o \) is open, or it is closed and \( o.\text{inv} \) holds in \( h \) by the induction hypothesis, so rule 4 applies. Either way, (G1) holds in \( h' \) for \( o \).

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. SC satisfies the hypotheses of Lem. 1; allocation satisfies rule 2; unwrapping satisfies rule 2 and wrapping satisfies rules 2 and 4 (we assume that first updates the \( \text{owner} \) attribute of every object in the \( \text{owns} \) set of its argument, and then updates the \( \text{closed} \) attribute of its argument); remember that \( \text{closed} \) and \( \text{owner} \) are only changed by wrap and unwrap. Attribute update satisfies rule 4.

It also satisfies the hypotheses of Lem. 2; allocation satisfies rule 5; invariant admissibility and the rules of language syntax satisfy rule 4. Rule (4) requires more details. First note that invariant admissibility requires that no invariable mention \( \text{owner} \); thus no object is concerned with wrapping (the only operation that can change \( \text{owner} \)), which therefore vacuously satisfies rule 4. Now, consider an update \( x.a := y \) with \( a \neq \text{owner} \) and \( a \neq \text{closed} \), and let \( o \) be any concerned object. Assuming \( o.\text{closed} \) and \( o.\text{inv} \) hold for a generic heap \( h \), we have to show that \( o.\text{inv} \) also holds of the heap \( h' = h[x.a \mapsto y] \). By definition of read set, \( x \in \text{read}(o.\text{inv}) \); \( o.\text{inv} \) is also admissible and hence it satisfies (A1); therefore \( x \in \{ o \} \cup o.\text{owns} \cup o.\text{subjects} \). However, the first precondition of the attribute update rule says that \( x \) is open; thus \( x \neq o \) because \( o \) is closed. We already proved that \( h \) satisfies (A2); for \( p = o \) this entails that all objects in \( o.\text{owns} \) are closed; therefore, \( x \notin o.\text{owns} \) as well. We conclude that \( x \in o.\text{subjects} \), which, combined with condition (A2) for \( o.\text{inv} \)’s admissibility, implies that \( o \in x.\text{observers} \) holds in \( h \). Finally, the second precondition of the attribute update rule establishes guard\((x.a := y, o)\), and thus by admissibility condition (A3), \( o.\text{inv} \) still holds in in the heap \( h' \).

As a closing remark, we note that another way to show soundness of SC is via reduction to LCI. To encode collaboration in LCI on top of the ownership encoding detailed in [5], we add the following clauses to the invariant of each class: one stating that all \( \text{subjects} \) know \( \text{Current} \) for an observer (the consequent of (A3), and for each
attribute of `current`, another one stating that all `observers` approve of the changes to this attribute.

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 short-hands `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 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`.

Note that we had to restate the first invariant clause of `OBSERVER` from Fig. 1 in terms of `observers` instead of `subscribers`. In general, collaboration invariants have to be expressed directly in terms of attributes of subjects and cannot refer to their ownership domains (including through logical functions). This is not a syntactic restriction but follows from the fact that it is rarely possible to establish a subject/observer relation with the whole domain (in this example, we would have to require `LIST` to allow `OBSERVER` objects in its `observers` set). This limitation can always be easily circumvented, however, by introducing a ghost attribute in the subject that mirrors the requires state.

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

make(v: INTEGER) -- Constructor
require open
modify Current
do
value := v
create subscribers
owns := [subscribers]
wrap
ensure
subscribers.is_empty
wrapped
end

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
value := v
wrapped
all o ∈ observers: o.wrapped
observers := old observers
end

feature [OBSERVER]
register(o: OBSERVER)
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]
subjects = []
end

class OBSERVER
subject: SUBJECT
cache: INTEGER

make(s: SUBJECT) -- Constructor
require
open
s.wrapped
modify Current, s
do
subject := s
s.register(current)
cache := s.value
subjects := [s]
wrap
ensure
subject := s
wrapped
s.wrapped
end

feature [SUBJECT]
notify
require
open
subjects := {subject}
s.notifyObservers has(current)
s.observers := []
modify Current
do
cache := subject.value
ensure
inv
end

invariant
cache = subject.value
subjects := [subject]
s.notifyObservers has(current)
observers := []
owns := []
end

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

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

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 the following defaults.

Pre- and postconditions: public procedures (methods not returning values) require and ensure that Current, its observers, and method arguments be wrapped.

Modify clauses: procedures modify Current; functions (methods returning values) modify nothing.

```java
class COLLECTION [G]

count: INTEGER

make (capacity: INTEGER) -- Constructor
require
open
capacity ≥ 0
modify Current
do
create elements(1, capacity)
ensure

elements.count = capacity
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

count = old.count − 1
wrapped
observers = {}
all o ∈ old.observers: o.open

feature {ITERATOR}
elements: ARRAY [G]

invariant
0 ≤ count and count ≤ elements.count
owns := [ elements ]
subjects := {}
end

class ITERATOR [G]

target: COLLECTION [G]
before, after: BOOLEAN

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

feature {NONE}
index: INTEGER

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
```
Invariants: Built-in ghost set attributes (such as `owns`) are invariably empty if they are not mentioned in the programmer-written invariant.

Wrapping: public procedures start by unwrapping `Current` and terminate after wrapping it back.

Built-in set manipulation: if a built-in ghost set attribute `s` is only mentioned in an invariant clause of the form `s = expr`, then `s` is considered implicit; correspondingly, every `wrap` of objects enclosing `s` will implicitly perform an assignment `s := expr`.8

These defaults encourage considerate programming: unless explicitly specified otherwise, an object is always required to restore the consistency of its observers at the end of a public method. This is a useful property, since the considerate paradigm promotes encapsulation and is convenient for the clients. Nevertheless, 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.

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 `guard(x.a := y, 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 `inv(o) ⇒ inv(o)_{x.a → 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 `guard(x.observers := y, o) = o ∈ 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, in particular those described in Sect. 3.

---

8 This is inspired by the default “static” treatment of `owns` sets in VCC.
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.

**Observer** [12,17,15] (see also SAVCBS ’07 [19], and Sect. 2).

The invariants of the observer objects depend on the state of the subject. Verification must ensure that the subject reports all its state changes to all observers, so that their clients can always rely on a globally consistent state. As additional challenge: combination with ownership (the subject keeps references to its observers in a collection, which is a part of its representation).

**Variants**: a simplified version where the number of observers is fixed (thus collections of observers are not needed); a more complex version with multiple observer classes related by inheritance, each class redefining class invariant and implementation of the notify method.

**Iterator** [12] (see also SAVCBS ’06 [19], and Sect. 2). Unlike observers in the Observer pattern, the implementation of a collection is not aware of the iterators operating on it. Specification must still be able to refer to the iterators attached to the collection while avoiding global reasoning. As additional challenge: we cannot rely on the implementation following considerate programming (where objects must be in consistent states at public call boundaries).

**Variants**: a more complex version where iterators may modify the collection.

**Master clock** [1,12]. 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** [11,14]. The specification expresses the consistency of the left and right neighbors directly attached to each node. Verification establishes that updates local to a node
Table 5: The challenge problems specified and verified using SC.

| PROBLEM   | SIZE (LOC) | CODE | REQ | AUX | SPEC/CODE | 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          |

(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** [23][22][9], (see also SAVCBS ’08 [19]). 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 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 whole-subtrees.

**PIP** [23][22]. The Priority Inheritance Protocol [21] 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 of update operations 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
Table 6: Comparison of invariant protocols on the challenge problems.

| Methodology | Observer | Considerate | Spec# | Friends | History | SC |
|-------------|----------|-------------|-------|---------|---------|----|
| VISIBLE-STATE SEMANTICS | Cooperation | + | + | + | + | + |
| | Iterator | _a | _a | + | + | + | + |
| | Master clock | _a | _a | + | + | + | + |
| | DLL | + | + | + | + | + | + |
| | Composite | _b | _b | _b | _b | _b | _b |
| | PIP | _b | _b | _b | _b | _b | _b |

_Only considerate programming_  _b Only bounded set of reachable subjects_  _c No framing specification_  _d No invariant stability_

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., (REQ + AUX)/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 [23]. 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=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.

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 approaches (focusing on those discussed in Sect. 3) on our six challenge problems. Tab. 4 reports how each methodology fares on each challenge problem: _−_ for “methodology not applicable”, _+_ for “applicable”, and _⊕_ 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. 4). Most approaches cannot deal with unbounded sets of subjects, and hence are inapplicable to Composite and PIP. The methodology of [22] 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 [15,22] 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 [11], 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 [11] 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 [11] 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 [12] support the same level of decoupling, but they cannot preserve stability with the reset method.

Reasoning without invariants. Other, more fundamental verification methodologies not based on invariants, such as dynamic frames [7] and separation logic [18], can fully handle all the six benchmark problems. The generality they achieve is, however, not without costs, as one loses stability of consistency properties (e.g., SUBJECT is not required to notify all its observers). Using recursive predicates instead of invariants to define global consistency also loses locality of specifications; for example, updates local to a node in a doubly-linked list require to reason about the whole list; and one node that becomes inconsistent during global updates in the Composite example makes the whole structure inconsistent (instead of just the parent). Recursive predicates over cyclic structures such as PIP also introduce non-trivial proof obligations to check they are well-founded.

SAVCBS workshops solutions. SC also fares favorably compared against the solutions submitted to the SAVCBS workshops [19] challenges (Iterator, Observer, and Composite). Considering only solutions for general-purpose languages and targeting complete requirement specifications, there are two solutions to the Iterator problem and two to the Composite problem. One solution to the Iterator uses JML and ESC/Java2; the collaborating parts of the invariants are, however, described by pre- and postconditions. One solution to the Composite also uses JML; it is hard to compare it to other solutions as it is based on model programs and proves invariant preservation only for methods that refine the model program used as specification. One solution to the Composite uses separation logic and VeriFast; the specification overhead for clients is higher than in our solution but there is no ghost state in the nodes (which has to be updated during global modifications), thus it has advantages and disadvantages compared to our solution.

6 Conclusions and Future Work

We presented semantic collaboration: a novel methodology for specifying and verifying invariants of arbitrary object structures. Compared to existing invariant protocols, it offers considerable flexibility and conceptual simplicity, as it introduces no ad hoc syntax and does not syntactically restrict the form of invariants. We implemented semantic
collaboration as part of the AutoProof Eiffel program verifier. Our experiments with six challenge problems demonstrate the wide applicability of the methodology.

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.

When it comes to reasoning about invariants, sequential and concurrent programs each have their distinctive challenges. In a sequential setting, one typically performs state updates in series of steps that temporarily break object consistency; this is acceptable since intermediate states are not visible to other objects. A sequential invariant protocol must adequately support such update schemes, while making sure that invariants hold at “crucial” points. Concurrent invariant protocols deal with different schemes, and hence have different goals. For this reason, we do not recommend extending SC to deal with concurrent programs; rather, it could be combined with an invariant protocol for concurrent programs, as done in VCC.

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. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: FM. pp. 268–283 (2006)
8. 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)
9. 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)
10. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: FM. LNCS, vol. 5850, pp. 806–809 (2009)
11. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: ECOOP. pp. 491–516 (2004)
12. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: ESOP. pp. 80–94 (2007)
13. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 2nd edn. (1997)
14. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Cooperation-based invariants for OO languages. Electr. Notes Theor. Comput. Sci. 160, 225–237 (2006)
15. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Invariants for non-hierarchical object structures. Electr. Notes Theor. Comput. Sci. 195, 211–229 (2008)
16. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. 62(3), 253–286 (2006)
17. Parkinson, M.J.: Class invariants: the end of the road? In: IWACO. ACM (2007)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74 (2002)
19. SAVCBS workshop series. http://www.eecs.ucf.edu/~leavens/SAVCBS/ (2001–2010)
20. Semantic Collaboration website. http://se.inf.ethz.ch/people/polikarpova/sc/
21. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput. 39(9), 1175–1185 (1990)
22. Summers, A.J., Drossopoulou, S.: Considerate reasoning and the composite design pattern. In: VMCAI. pp. 328–344 (2010)
23. Summers, A.J., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: IWACO. pp. 1–9. ACM (2009)