A Logical Account of Hoare’s Mismatch
Information Hiding via Second Order Framing in Region Logic

Anindya Banerjee
IMDEA Software, Madrid, Spain
anindya.banerjee@imdea.org

David A. Naumann
Stevens Institute of Technology, Hoboken NJ, USA
naumann@cs.stevens.edu

Abstract
We investigate information hiding in object-based programs and the associated mismatch. While client reasoning is in terms of interface specifications, the implementation of an interface is verified against different specifications that involve invariants about internal data structures. Soundness of this mismatched reasoning depends on encapsulation of internal data structures. The problem is that encapsulation is notoriously difficult to achieve in contemporary software in which shared mutable objects are ubiquitous. We account for the mismatch via proof rules that phrase the mismatch using explicit conditions that are imposed on client effects. Effects are tracked using ghost state and separation assertions in a style that has been used in a number of verification tools. Our approach permits the formulation of encapsulation disciplines (such as ownership, or package confinement) as part of the interface specification, in the form of a dynamic boundary, rather than as a discipline directly baked into a verifier. One implication is that we can provide a foundation for the axiomatic semantics of these verifiers. Our approach is flexible in that disciplines can be used on a per-module basis and then combined to achieve end to end soundness.

1. Introduction
Many programs manipulate shared, mutable objects, be they memory segments in C code, heap records in ML code, or objects in the sense of Java-like languages. Most software designs are based on abstraction at many scales. For example: A Point offers operations such as repositioning without revealing whether the internal representation uses polar or cartesian coordinates. A Collection offers addition and removal of elements without revealing whether the internal representation is a balanced binary tree or something else. An application framework (e.g., the Google Web Toolkit) offers complex concepts and facilities without revealing the still more complex underlying infrastructure. Abstraction is achieved by hiding irrelevant details, in particular internal data and invariants on which the implementation relies. Hiding of internal invariants means that they do not appear in specifications used by “clients” of the abstraction. This mismatch between what is verified about implementations and what is assumed by clients was articulated and justified by Hoare: invariants should depend only on data that is encapsulated to prevent clients from falsifying the invariants [17].

It is notoriously difficult to achieve encapsulation in the presence of shared, dynamically allocated mutable objects [20, 27]. Current tools for software verification either do not support hiding of invariants (e.g., Jahob [35], jStar [12]), do not treat object invariants soundly (e.g., ESC/Java [14], Eiffel) or at best offer soundness for restricted situations where hierarchical structure can be imposed on the heap (e.g. Spec# [4]) which is the best that is provided by current theory.

Scoping does provide some encapsulation. Class Point declares its fields as private, i.e., visible only to code within the class. Classes pertinent to collections — Set, List, iterators, various data structures— may be grouped together in a module. Scoping mechanisms are essential but insufficient for reasoning about dynamically allocated mutable objects.

Building on scope and type mechanisms it is possible to enforce module level alias control (as in Confined Types [15]) to justify hiding of invariants that depend on some or all instances of some classes declared in the module. For encapsulation at the granularity of instance-oriented “object invariants”, various ownership disciplines have been introduced [9, 7, 11, 25, 23]. Some have been deployed in verification systems, e.g., in JML tools [8] and in Spec# [4]. For example, suppose for simplicity that an instance of class Set is viewed as owning the nodes of a linked list and a type system enforces that neither clients nor other instances of Set can access these nodes. Code in Set relies on an invariant, such as the absence of duplicate elements, that can be hidden from clients. The inflexibility of various formalizations of ownership has led to quite a few competing and incompatible variations, each described in a way that requires its global imposition on all program components (e.g., [9, 7, 11]). Another problem is that, although it is easy to embody Hoare’s mismatch in the axiomatic semantics used by a verifier, the details are not often justified rigorously.

Separation Logic [28] (SL) offers elegant means to reason about the footprint of an invariant, i.e., the heap locations on which it depends, and the footprint of a command, i.e., the locations possibly written. On this basis, O’Hearn et al [27] give what we call a second order frame rule (SOF) that directly embodies Hoare’s mismatch. By contrast with ownership disciplines, the SOF rule pertains directly to hiding on the granularity of a module (i.e., a group of procedures sharing some encapsulated state). We show that this flexibly encompasses invariants such as those that involve cooperating clusters of objects such as a Subject and its Observers. Soundness of the SOF rule has been shown directly in terms of a standard, non-axiomatic program semantics [29]. The rule relies on two critical features of SL. The separating conjunction $P * Q$ expresses that formulas $P$ and $Q$ are both true, of disjoint parts of the heap. The tight interpretation of a correctness judgement $\{P\} C\{ Q\}[\tau]$ says that $C$ is not only correct but neither reads nor writes outside the footprint of $P$. The Hoare triple is augmented by a modifies clause that lists the variables $\tau$ that may be written.

Our contribution is a SOF rule for a classical first-order assertion language that uses an ordinary interpretation of correctness judgements (for fault-avoiding partial correctness). This is achieved by making footprints explicit using regions (sets of references), including expressions for the image of a region under a field name and for disjointness of regions. State-dependent region expressions are used in the modifies clause and in a subsidiary judgement about footprints of formulas. Read and write footprints are expressed us-
ing (mutable) ghost state which avoids the need to express footprints using inductively defined predicates that traverse data structures.

**Contributions**

- We augment module interface specifications by including a *dynamic encapsulation boundary* which must be respected by clients. The dynamic boundary is described via read effects that approximate, in a way suitable to appear in the interface, the footprint of the hidden invariant.
- We extend *region logic* (RL) [3] to include procedures and correctness judgements with hypotheses (Sec. 3 and 4). We improve the treatment of hiding sketched briefly in that paper. On this basis we give our SOF rule (Sec. 5). We prove its soundness in a straightforward operational semantics, which validates standard proof rules, such as Hoare’s rule of Conjunction, and can use a deterministic or nondeterministic memory allocator.
- By contrast with SL, (a) our SOF rule allows the hidden invariant to depend on state that is also read by client programs (e.g., global entities in an application framework); and (b) it allows invariants that are not precise [27], i.e., they do not have a unique minimal footprint. Precise predicates typically traverse a data structure (e.g., “every reachable node is balanced”) whereas imprecise ones involve existential quantification (e.g., “some queue is nonempty”). In Sec. 7 we examine why our setting is not susceptible to Reynolds’ conundrum [27] which shows that in SL the SOF rule needs to be restricted to precise invariants or precise specifications to be consistent with the rule of Conjunction.
- We show by examples (Sec. 2 and 6) that our SOF rule can hide invariants that pertain to several objects with a single owner, as well as design patterns in which several peers cooperate (which are incompatible with ownership and remain as challenge problems in the current literature [20, 6, 19]). A program may link together multiple modules, each with its own hidden invariant and dynamic boundary. Our approach encompasses alias confinement disciplines that are enforceable by static analysis [11] as well as less restrictive disciplines that impose proof obligations on clients as in ownership transfers that are “in the eye of the asserter” [27].

Our explicit use of ghost state for frame specifications and separation reasoning was directly inspired by the *dynamic frames* of Kassios [18], whence our term “dynamic boundary”. Similar uses of ghost state, including pure first order encodings of reachability properties, have been found effective in verification tools based on automated theorem provers [16, 10, 35]. One of our aims is to justify the axiomatic semantics used in such tools, which often embody Hoare’s mismatch. We pay a price in verbosity compared with SL, in that ghost fields and variables need to be declared, assigned, and used in specifications. For example, the code of class Set in Sec. 2 assigns newly allocated list nodes to a “rep” field, and the code in our memory manager example (Sec. 6) updates a region variable holding the objects currently “owned” by the manager.

For readability, the formalization (Sec. 3 and 4) is for barebones programs with rudimentary procedures. Related work is discussed further in Sec. 7.

Our aim is not to advocate a particular proof system (and certainly not a module system) but rather to move beyond competing attempts to provide the “right” specification notation and programming discipline for hiding invariants on mutable state. We provide a logical bridge between internal and external views of a module interface, in which the encapsulation boundary is fully described through dynamic framing (to augment the conventional syntactic signature of the interface). On this basis we hope to shift attention to creating a corpus of specification patterns, with a range of granularity, generality, and balance of proof obligations between modules and their clients. Modular verification tools will eventually be able to support integrated use of complementary disciplines for a wide range of program design patterns and application frameworks.

## 2. Synopses

For a first example of second order framing, we consider the simple program in Fig. 1.1 This section begins by introducing region logic as used by the interface specifications in Fig 2. Then we consider reasoning about the implementation (Fig. 1) using an invariant, which serves to illustrate local reasoning via a first order “frame rule” akin to Hoare’s rule of Invariance. Finally, we consider reasoning about a client program, using the interface specifications. To justify hiding the invariant, the interface includes a dynamic boundary for the invariant and the client code is obliged not to write within that boundary. We conclude by sketching how soundness of this reasoning is captured by the SOF rule.

![Image](image1.png)

**Figure 1.** Library code for Set example. Class Set and variable pool comprise a module.

<table>
<thead>
<tr>
<th>Method</th>
<th>Pre-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set()</td>
<td>elements = empty ∧ pool = old(pool)∪{self}</td>
</tr>
<tr>
<td>add(i: int)</td>
<td>elements = old(elements)∪{i}</td>
</tr>
<tr>
<td>contains(i: int)</td>
<td>result = (i ∈ elements)</td>
</tr>
<tr>
<td>remove(i: int)</td>
<td>elements = old(elements) – {i}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set()</td>
<td>wr pool</td>
</tr>
<tr>
<td>add(i: int)</td>
<td>wr {self}∗any, wr alloc</td>
</tr>
<tr>
<td>contains(i: int)</td>
<td>(none)</td>
</tr>
<tr>
<td>remove(i: int)</td>
<td>wr {self}∗any, wr pool∗rep∗any</td>
</tr>
</tbody>
</table>

**Figure 2.** Public specifications for class Set. Preconditions: true.

```java
var s : Set = new Set(); var n : Node = new Node();
var s.add(1); s.add(2); n.val = 1; s.remove(1);
b = s.contains(1);
```

**Figure 3.** Example client, in context of variable b : boolean.

and their clients. Modular verification tools will eventually be able to support integrated use of complementary disciplines for a wide range of program design patterns and application frameworks.

1The programming notation is similar to Java, in particular a value of a class type like Node is either null or a reference to an allocated object with the fields declared in the class. Methods have an implicit parameter, self.
The specifications (Fig. 2) are expressed in terms of an integer set, \( \textit{elements} \), that could be formalized as a “model field”. Abstraction of this sort is commonplace and important, but not the focus of this paper so we don’t formalize \( \textit{elements} \).

What is important is that we shall add to the interface a dynamic boundary which abstracts from the state to be encapsulated for a hidden invariant. To this end, the implementation in Fig. 1 is instrumented using ghost state of type \( \textit{rgn} \) for a hidden invariant. To this end, the implementation in Fig. 1 is instrumented using ghost state of type \( \textit{rgn} \); a \textit{region} is a set of allocated references of any type, possibly also containing null. Inclusion of null is one of several small deviations from our paper [3], which remains useful for a more detailed introduction to the basic logic.

The effects clause in Fig. 2 for the constructor, \( \textit{Set}() \), is a conventional modifies specification that says variable \( \textit{pool} \) is allowed to be assigned. Our chosen notation and terminology reflects a feature that deals with framing issues beyond the scope of the paper, namely, that both read and write effects may be ascribed to commands and procedures.

For \( \textit{add} \), the write effect \( w \{ \textit{self} \} \) any is also a conventional one that says fields of self may be written (cf. \( \textit{self} . \textit{state} \) in a JML or Spec# modifies clause). The expression \( \{ \textit{self} \} \) denotes a singleton \textit{region} containing the value of \textit{self} (an object reference). For any field name \( f \) and region expression \( G \), the effect \( w G^f \) allows update of the \( f \) fields of objects in \( G \) (and is read “write \( G \) image \( f \)”). The special name “any” abstracts from specific field names to allow any field to be written; this can be refined to “data groups” [24, 8] that abstract from specific field names that should not be exposed to clients.

The primitive region expression “\( \textit{alloc} \)” allocates the current domain of the heap, i.e., the set of allocated references. The effect of \( \textit{add} \) includes \( \textit{w alloc} \) which means new objects may be allocated.

For \( \textit{remove} \), the most interesting effect is \( w \textit{pool}^t \textit{rep} \) any. Here \( \textit{pool}^t \textit{rep} \) is a region expression; it denotes the union of the \( \textit{rep} \) fields of all objects in \( \textit{pool} \). The effect says fields of objects in \( \textit{pool}^t \textit{rep} \) (in the initial state) may be written. For any \( s \) of type \( \textit{Set} \), field \( \textit{rep} \) is intended to hold the (references to) the nodes reached from \( s . \textit{lst} \), and \( \textit{pool}^t \textit{rep} \) is the union of those regions, i.e., all the nodes used by \( \textit{Sets} \) in \( \textit{pool} \). (To allow for alternate implementations it might be wise for \( \textit{add} \) to also have the effect \( w \textit{pool}^t \textit{rep} \) any.)

The effect \( w \textit{pool}^t \textit{rep} \) any is a “dynamic frame” because \( \textit{rep} \) is a mutable field and \( \textit{pool} \) a mutable variable: the meaning of the effect is state dependent.

\textbf{A module invariant.} For efficiency, our implementation of \( \textit{remove} \) relies on the invariant that no integer value is duplicated in the list rooted at \( \textit{lst} \). This invariant must be established by the constructor and preserved by all methods of class \( \textit{Set} \). However, as per what we call \textit{Hoare’s mismatch} [17], the invariant does not appear in the interface specifications as viewed by clients.

The invariant “no duplicates” pertains to a single instance \( s \) of \( \textit{Set} \), together with its list. The ghost field \( s . \textit{rep} \) is intended to refer to the set of nodes reachable from field \( s . \textit{lst} \). To avoid the need for reachability, which is costly for automated verification and not available in all assertion languages, we approximate it using the invariant that \( s . \textit{rep} \) contains only nodes “owned” by \( s \). Consider the following condition. The first conjunct says there are no duplicates. The next two say that \( s . \textit{rep} \) is \( n \textit{xt}-closed \) and contains \( s . \textit{lst} \). The last says all nodes in \( s . \textit{rep} \) are owned by \( s \).

\[ \text{SetI}(s : \textit{Set}) : \{ \forall n : \textit{Node} \in s . \textit{rep} | \; n = m \lor \text{null} \neq m . \textit{val} \} \land \{ s . \textit{lst} \in s . \textit{rep} \land \{ s . \textit{rep} \cdot \textit{nxt} \subseteq s . \textit{rep} \land \{ s . \textit{rep} \cdot \textit{own} \subseteq \{ \} \} \}
\]

Note that “\( n \in s . \textit{rep} \)” is considered false in case \( s = \text{null} \).

Were we to extend the example by adding iterators, we might find that the natural granularity for an invariant would be a \( \textit{Set} \) together with its iterators as well as its list. To avoid commitment to invariants of a specific granularity, we consider \textit{module invariants}, associated with the program syntax and unit of scope: a “module”, consisting of one or more class declarations. For example, the module invariant could say \( \textit{SetI} \) holds for every instance of \( \textit{Set} \). To illustrate flexibility in choosing encapsulation boundaries, we choose instead to say \( \textit{SetI} \) holds for only those instances that are in \( \textit{pool} \) (which, e.g., could be those returned by a factory method):

\[ \text{I} : \; \text{null} \notin \text{pool} \land \forall s : \text{Set} \in \text{pool} | \; \text{SetI}(s) \]

We shall find a “frame” or footprint for \( \textit{I} \), that can serve as a dynamic boundary expressing the state-dependent aspect of the encapsulation that will allow \( \textit{I} \) to be hidden from clients. But first we seek a frame for the object invariant \( \textit{SetI}(s) \), which will be used for “local reasoning” [28] at the granularity of a single instance of \( \textit{Set} \). The frame is given by read effects that describe the part of the state on which the value of \( \textit{SetI}(s) \) may depend:

\[ \exists \_ : \; \text{null} \notin \text{pool} \land \forall s : \text{Set} \in \text{pool} | \; \text{SetI}(s) \]

A region logic provides rules for a judgement that says certain read effects bound the footprint of a formula, in this case:

\[ \text{true} \vdash \exists \_ \text{ from } \text{SetI}(s) \]

(1)

The effect \( r \) says \( \textit{SetI}(s) \) may depend on variable \( s \) and \( \{ s \}^t \textit{rep} \cdot \textit{lst} \) says it may depend on \( s . \textit{rep} \) and \( s . \textit{lst} \). The dynamic part is \( r d \{ s . \textit{rep} \cdot \textit{nxt}, \textit{val}, \textit{own} \} \) which says it depends on the \( \textit{nxt}, \textit{val}, \textit{own} \) fields of some objects in \( s . \textit{rep} \). The judgement (1) involves a formula, here \textit{true}, because effects can involve region expressions like \( s . \textit{rep} \) that depend on mutable state, so framing relationships may hold only under some conditions on that state. For example, we can derive

\[ s \in \text{pool} \vdash r d \text{pool}^t \cdot \textit{rep} \cdot \textit{lst} \text{ from } s . \textit{lst} \in s . \textit{rep} \]

(2)

Using judgements including (2) and (1) we can derive a frame judgement true \( \vdash \exists \_ \text{ from } I \) for the module invariant, where

\[ \exists I : \; r d \text{ pool}, \; \text{pool}^t \cdot \textit{rep} \cdot \textit{lst}, \; \text{pool}^t \cdot \textit{rep} \cdot \textit{nxt}, \textit{val}, \textit{own} \]

An abstraction of \( \exists I \) will be used later, when we turn to verification of the client.

\textbf{Framing for local reasoning using the invariant.} For the implementations in Fig. 1, we would like to reason “locally” in terms of a single \( \textit{Set} \). The ownership conditions in \( \textit{SetI}(s) \) yield an “island confinement” property (where \( \# \) denotes disjointness of sets):

\[ I \Rightarrow (\forall s, t : \textit{Set} \in \text{pool} | \; s = t \lor \{ s \}^t \textit{rep} \# \{ t \}^s \textit{rep} ) \]

(3)

because if \( s \neq t, n \neq \text{null}, \) and \( n \) is in \( s . \textit{rep} \cap t . \textit{rep} \) then \( n . \textit{own} = s \) and \( n . \textit{own} = t \), a contradiction. To verify that method \( \textit{add} \) preserves \( I \), we exploit island confinement in order to focus on the object invariant. Now, \( I \) is equivalent to \( \textit{SetI}(\textit{self}) \land \textit{Iexcept} \), where

\[ \textit{Iexcept} : \; \text{null} \notin \text{pool} \land \forall s \in \text{pool} \neg \{ \textit{self} \} \land \text{SetI}(s) \]

We can frame \( \textit{Iexcept} \) by the effects

\[ \]
The footprint of Iexcept is disjoint from the footprint of SetI (self). More to the point, let Badd be the body of method add. By ordinary means we can verify the following Hoare triple:

\[ \{ \text{SetI}(\text{self}) \} B\text{add} \{ \text{SetI}(\text{self}) \wedge \text{elements} = \text{old}(\text{elements}) \cup \{ i \} \} \]

Next, we exploit separation to conjoin Iexcept to the pre and post conditions because the write effects of add (Fig. 2) are separate from the read effect of Iexcept. To make this precise, we define an operator: If \( \delta \) is a set of read effects and \( \pi \) is a set of write effects then \( \delta \ast \pi \) is a conjunction of disjointness formulas, validity of which ensures that writes allowed by \( \pi \) cannot affect the value of a formula with footprint \( \delta \). (The formula \( \delta \ast \pi \) is defined by induction on the syntax of the effects.) Here is our first order frame rule (from [3]):

\[
\text{FRAME} \quad \vdash \{ P \} C \{ P' \} [\pi] \quad P \vdash \delta \text{ from } Q \quad P \Rightarrow \delta \ast \pi
\]

It happens that \( \delta_1 \ast (\text{wr} (\text{self}) \ast \text{any}, \text{wr alloc}) \) is true, so we can take \( Q \) to be Iexcept in rule FRAME to complete the proof of \( \{ I \} B\text{add} \{ I \wedge \text{elements} = \text{old}(\text{elements}) \cup \{ i \} \} \).

**Reasoning about a client while hiding the invariant.** Besides verification of the module’s method bodies with respect to specifications in which the invariant \( I \) is explicit, there is another obligation on the module. It must declare a dynamic boundary that frames \( I \). It would not be appropriate to use \( \delta_1 \), which mentions field \( \text{lst} \) that would be declared private. We choose to use \( \bar{\theta}_1 : \text{rd pool, pool}^*\text{any, pool}^*\text{rep}^*\text{any} \)

The obligation is \( I \vdash \bar{\theta}_1 \text{ from } I \), which is derivable from true \( \vdash I \text{ from } I \) by a subsumption rule.

Something is needed to ensure that \( I \) is initially true. Typical formalizations include an initializer command, so the client program takes the form let \( m \) be \( B \) in \( (\text{init}; C) \). With dynamic allocation, it is constructors that do much of the work to establish invariants. In the present example, let us define \text{Init} to be the condition \( \text{pool} = \varnothing \) which is suitable to be declared in the module interface. Note that \text{Init} \( \Rightarrow I \) is valid.

Finally, we can turn to verifying the simple client command in Fig. 3. We will verify that under precondition \text{Init} the client establishes postcondition \( b = \text{false} \). Here is a proof outline.

\[
\{ \text{Init} \}
\]

\[
s := \text{new } \text{Set}();
\]

\[
\{ \text{s.elements} = \varnothing \wedge \text{pool} = \{ s \} \} \quad \text{by spec of Set}
\]

\[
n := \text{new } \text{Node}();
\]

\[
\{ \text{s.add}(1); \text{s.add}(2); \}
\]

\[
\{ \text{s.elements} = \{ 1, 2 \} \wedge \text{pool} = \{ s \} \} \quad \text{by spec of add}
\]

\[
n.\text{val} := 1;
\]

\[
\{ \text{s.elements} = \{ 2 \} \wedge \text{pool} = \{ s \} \} \quad \text{by spec of remove}
\]

\[
b := \text{s.contains}(1);
\]

\[
\{ b = \text{false} \} \quad \text{by spec of contains}
\]

As it should, this reasoning uses the interface specifications (Fig. 2). However, there is an additional proof obligation. For it to be sound to hide \( I \) from the client, we need that the client’s write effects are separate from the dynamic bound \( \bar{\theta}_1 \) — not just pre/post effects but also effects at intermediate steps, at which module methods like add are called. We include the assignment \( n.\text{val} := 1 \) as a simple example of how encapsulation might be violated (but is not). If this assignment was replaced by \( s.\text{lst}.\text{val} := 1 \) then indeed it would break the invariant and render the program incorrect.

The effect of \( n.\text{val} := 1 \) is \( \text{wr} (\{ n \} \ast \text{val} \) and it must be shown to be outside the boundary \( \bar{\theta}_1 \). After all, \( I \) reads field \( \text{val} \) as seen in the more precise footprint, \( \delta_1 \), subsumed by \( \bar{\theta}_1 \). By definition of \( \ast \), we have that \( \bar{\theta}_1 \ast \text{wr} (\{ n \} \ast \text{val} \) is \( \{ n \} \ast \text{pool} \wedge \{ n \} \ast \text{pool} \ast \text{rep} \). We must show it holds just before the assignment \( n.\text{val} := 1 \).

The condition \( \{ n \} \ast \text{pool} \) is equivalent to \( n \not\in \text{pool} \) which clearly holds — none of the client code writes pool and the value of \( n \) is fresh.\(^4\) There are several ways to show \( \{ n \} \ast \text{pool} \ast \text{rep} \). One way is to notice that a form of “package confinement” [15] applies here: references to the instances of \text{Node} used by the Set implementation are never made available to client code. (That is stronger than the property required here, which is merely that such references are not misused by clients.)

To be very explicit about the reasoning that could be embodied by a static analysis for conformance, we shall exploit field \( \text{own} \) of class \text{Node}. The analysis might ensure the all-states invariant that any \text{Node} pointer \( p \) available to the client code has \( p.\text{own} = \text{null} \) (the default from \text{Node’s} constructor). Moreover, nodes are not “leaked” by the module code, which can be shown using

\[
\text{R} : \quad \text{pool}^*\text{rep}^*\text{own} \subseteq \text{pool} \wedge \text{null} \not\in \text{pool}
\]

Unlike \( I \), formula \( R \) is suitable to appear in the interface. A general fact about region images is that \( G \ast f \subseteq H \) and \( x.f \not\in H \) imply \( x \not\in G \) (where \( G, H \) are any region expressions and \( f \) any field). So using \( R \) we get \( n \not\in \text{pool}^*\text{rep} \) as required by the obligation to respect the dynamic encapsulation boundary.

A dynamic boundary is expressed in terms of state potentially mutated by the module implementation, e.g., the effect of add in Fig. 1 allows writing state on which \( \bar{\theta}_1 \) depends.\(^5\) So interface specifications need to provide clients with sufficient information to reason about the boundary. In the example, \( R \) could be explicitly conjoined with the interface’s method specifications, or declared as a public invariant [21]. In our memory manager example (Sec 6), it is not a fixed invariant but rather the individual method specifications that allow clients to reason about the boundary.\(^6\)

In summary, the verification of our client and its module are justified by the following rule which embodies Hoare’s mismatch:

\[
\Delta(\delta) ; \Theta(\delta) \vdash \{ P \} C \{ P' \} [\pi]
\]

\[
\Delta(\delta) ; (\Theta \otimes I) ; \vdash \{ Q \wedge I \} B \{ Q' \wedge I \} [\pi]
\]

\[
I \vdash \bar{\theta} \text{ from } I \quad \text{Init} \Rightarrow I
\]

\[
\Delta(\delta) \vdash \{ P \wedge \text{Init} \} \text{ let } m \text{ be } B \in C \{ P' \} [\pi]
\]

Here for simplicity \( \Theta \) consists of a single method specification, \( \{ Q \} \{ \pi \} [\pi] \). The first antecedent expresses the proof obligation on the client program \( C \): in addition to the usual pre/post/modifies specification, it must respect the dynamic boundary \( \bar{\theta} \) which (please note!) is written together with the specification of \( m \). In general, a boundary is associated with each group of methods taken to comprise the interface of a module, e.g., here we include a second module interface \( \Delta(\delta) \). To respect the boundary means that every primitive action within \( C \) must stay outside \( \bar{\theta} \) and \( \delta \). It is not enough that the end-to-end effect \( \pi \) of \( C \) is outside \( \bar{\theta}, \delta \), as \( \pi \) only describes effects on objects that exist in the initial state. Moreover, \( \pi \) includes

\(^4\) Had we chosen to use typed regions, or to maintain an invariant that every element of pool is of type Set, then \( n \not\in \text{pool} \) would hold because the type \text{Node} of \( n \) is not a subtype of Set.

\(^5\) Dynamic framing raises the issue of interference with effects, which is handled in the rule for sequential composition in [3].

\(^6\) Nonetheless, we believe the present situation, where \( I \Rightarrow R \), is typical. It suggests that the module interface specification may include, in addition to the dynamic bound, a public invariant that is framed by the bound and can be seen as part of the encapsulation discipline. In this case clients are not responsible for maintaining \( R \) because (from \( I \Rightarrow R \)) it is protected by the bound they are already required to respect. Nor is there an additional proof obligation on the module implementation.
effects from calls to methods of Θ and those effects are not required to be outside the boundary $\bar{\theta}$.

The second antecedent is the proof obligation on the implementation, $B$, of $m$, in which the invariant $I$ is explicit; the dynamic boundary for $Θ$ is empty, so $\bar{\theta}$ is not imposed on $B$, but the bound $\bar{\theta}$ on the ambient library must be respected by $B$. Any recursive calls to $m$ in $B$ are verified using the specifications $Θ \otimes I$ which means $I$ is conjoined, here $(Q \land \exists m)\{Q' \land I'[\eta]\}$. The side conditions $I \vdash \bar{\theta} \text{frm } I$ and $\text{Init } \Rightarrow I$ are obligations on the module implementation.

The rule above is a derived rule. Fig. 4 gives a generic derivation, using an ordinary rule that links a procedure to its implementation without any mismatch in specifications, together with the SOF rule and a rule to forget the dynamic boundary once it has served its purpose. The beauty of the SOF, the form of which is due to O’Hearn et al. [28], is that it distills the essence of Hoare’s mismatch —which, with proper encapsulation, is an intricate match.

3. Background: Region logic without procedures

Programming language. Fig. 3 gives the programming language syntax, except for procedures. The semantics is standard and deferred to Sec. 4. A program consists of a command $C$ in the context of some class declarations. Commands are standard: assignment, object allocation, field access, field update, conditionals, loops, and local variable blocks.

A class declaration class $K \{ T f \}$ introduces a type name $K$. The values of this type are null and all allocated references to mutable objects of type $K$ with fields $T$. Here and throughout, identifiers with an overline range over lists. In addition to int and reference types, there is type $r$gn with values ranging over finite sets of references of any type.

Program expressions, $E$, do not depend on the heap; $y.f$ is not an expression but rather part of the primitive field access command, $x := y.f$, for reading a field, as in SL. Region expressions, $G$, cannot influence control flow or the value of non-region fields/variables (as no integer expressions have subexpressions of type $r$gn); their purpose is to serve as ghosts for reasoning.

Typing. There is an ambient class table comprising a well formed collection of class declarations. We write fields($K$) for the field declarations $f : T$ of class $K$. Judgement $Γ ⊢ F : T$ says that in context $Γ$ that assigns types to variable names, region or program expression $F$ has type $T$, and $Γ \vdash C$ says $C$ is a well formed command in $Γ$. Type int is separated from reference types: there is no pointer arithmetic. The typing rule for singleton region $\{E\}$ enforces that $E$ is of reference type. The rule for region dereference, $G.f$, checks that $f$ is declared in some class $K$ and either $f$ is a field of class type, $K'$, or $f$ is of type $r$gn. The primitive command for field access, $x := y.f$, is restricted to non-region fields. For fields of type $r$gn, note that, e.g., $x := \{y\}f$ is permitted and is an instance of ordinary assignment, $x := F$. An implicit side condition on all typing and also proof rules is that both the consequent and the antecedents are well formed.

\[
x, y, r \in \text{VarName} \ f, g \in \text{FieldName} \ K \in \text{DeclaredClassNames}
\]

\[
T ::= \text{int} \mid K \mid rgn
\]

\[
E ::= x = c \mid c \mid \text{null} \mid E \oplus E \mid c \in Z \# \oplus c \in \{=, +, -, \ast, \ldots\}
\]

\[
G ::= x \mid \{E\} \mid \text{emp} \mid \text{alloc} \mid G+f \mid G \otimes G \oplus \in \{\cup, \cap, -\}
\]

\[
F ::= E \mid G
\]

\[
C ::= x := F \mid x := \text{new} K \mid x := y.f \mid x.f := F \mid \ldots
\]

Figure 5. Programming language (excerpts). Category $E$ is for program expressions and $G$ for region expressions.

\[
P ::= E = E \mid x.f = E \mid G \subseteq G \mid G \# G \mid \text{type}(K, G) \mid \forall z : \text{int} \mid P \mid \forall z : K : G \mid P \mid P \land P \mid \neg P
\]

\[
\sigma = x.f = E \quad \text{iff} \quad \sigma(x) \neq \text{null} \quad \text{and} \quad \sigma(x.f) = [E]\sigma
\]

\[
\sigma = \Gamma \otimes G \oplus \in \{\text{null}\}
\]

\[
\sigma = \Gamma \otimes x : K : G \mid P \quad \text{iff} \quad \text{extend}(\sigma, x, o) = \Gamma \otimes x : K : P \quad \text{for all} \quad o \in [G] \quad \sigma \text{s.t. } \sigma \neq \text{null} \quad \text{and} \quad \text{type}(o, \sigma) = K
\]

Figure 6. Formulas: grammar and semantics (excerpts).

Semantics. We assume given a set $\text{Ref}$ of reference values including a distinguished value, $\text{null}$. A $\Gamma$-state assigns values to the variables in $\Gamma$ and also has a heap. We abstract from the concrete representation of states and merely assume the following operations are available for a state $\sigma$: $\sigma(x)$ is the value of $x$, alloc($\sigma$) is the set of all allocated references, type($o, \sigma$) is the type of an allocated reference $o$, update($\sigma, o, f, v$) overrides $\sigma$ to map field $f$ of $o$ to $v$ (for $o \in \text{alloc}(\sigma)$), extend($\sigma, x, v$) extends $\sigma$ to map $x$ to value $v$ (for $x \notin \text{dom}(\sigma)$), new($\sigma, o, K, \mathbb{P}$) extends $\sigma$ to map fresh $o$ (i.e., assuming $o \notin \text{alloc}(\sigma)$) to a $K$-record with field values $\pi$ and type $K$. States are assumed to be well typed and to have no dangling references.

The semantics of program expressions $E$, written $[E]\sigma$, is straightforward and omitted. Note that $[E]\sigma$ is always a value (of appropriate type), never fault; moreover it only depends on the store, not the heap. For region expressions, the meaning of singleton region $\{E\}$ is $[[E]]\sigma$. The meaning of $\text{emp}$ is the empty set and that of alloc is $\text{alloc}(\sigma)$, i.e., the set of all allocated references. The meaning of $G.f$, if $f$ is a reference type, is the set containing the $f$-images of all non-null references in $G$'s denotation; but if $f : rgn$ then $G.f$ denotes the union of the $f$-images.

Assertions and effects. The assertion language appears in Fig. 6. Quantification is over int and reference types only (not over $r$gn). For reference types, a bounding region is required (e.g., region expression $G$ in $\forall x : K : G \mid P$ and the bound variable must not appear in the bound of the quantification). This facilitates framing. Fig. 6 also gives the semantics of a well formed formula, $\Gamma \vdash P$, as a satisfaction relation $\sigma \models \Gamma \vdash P$ that is defined for all $\Gamma$-states $\sigma$. A formula is valid iff it is true in all states.

Effects are given by the grammar

\[
e ::= r := x \mid r \text{d} G.f \mid w r x \mid w r G.f \mid r \text{d alloc} \mid r \text{d alloc} \mid fr G
\]

Effect $wr G.f$ says $f$ fields of any pre-existing object in $G$ may be written. Note that, e.g., $\text{in } wr \{x\}f$ the first apostrophe is forming a region expression $\{x\}f$ and then the second is indicating what field (of objects in the region) is allowed to be written. Effect $wr$ alloc allows allocation and $fr$ says that all elements of $G$ in the final state are freshly allocated. Read effect $rd G.f$ fields of any pre-existing object in $G$ may be read and $rd alloc$ allows reading the set alloc. An effect set is a comma-separated list of effects, ranged over by $\pi$ etc. Effects are well formed in $\Gamma$ if their constituent expressions are.

Freshness effects are needed to annihilate write effects of freshly allocated objects in sequences. Consider, e.g., $x := \text{new } K ; x.f := 0$; While the effect of the field update is $wr \{x\}f$, the pre-state of the entire sequence does not contain the freshly allocated object. Indeed, in this case no pre-existing objects are updated. See the sequence rule in [3].

Definition 1 (allows transition) Let effect set $\pi$ be well formed in $\Gamma$ and let $\sigma, \sigma'$ be $\Gamma$-states. We say $\pi \text{ allows transition from } \sigma \text{ to } \sigma'$, written $\sigma \rightsquigarrow \sigma' \models \pi$, if $\sigma'$ extends $\pi$ and the following all hold:

$\sigma'$ extends $\sigma$ provided $\text{alloc}(\sigma') \subseteq \text{alloc}(\sigma')$ and type($o, \sigma$) = type($o, \sigma'$) for all $o \in \text{alloc}(\sigma)$. 

\footnote{\begin{eqnarray*}
\text{extended version}
\end{eqnarray*}}
Figure 4. Schematic derivation for second order framing, assuming that Θ is a single specification \( \{ Q \} m(x : T) \{ Q' \} \). For SOF the side condition is \( I \vdash (\emptyset, rd \text{ alloc}) \) from \( I \) and for CONSEQ it is validity of \( \text{Init} \Rightarrow I \).

\[
\begin{align*}
&\Delta(\emptyset); (\emptyset \otimes I) \vdash \{ Q \wedge I \} B \{ Q' \wedge I \} [\pi] \\
&\Delta(\emptyset); (\emptyset \otimes I) \vdash \{ P \wedge I \} C \{ P' \wedge I \} [\pi] \\
&\Delta(\emptyset); (\emptyset \otimes I) \vdash \{ P \wedge I \} C \{ P' \wedge I \} [\pi] \\
&\Delta(\emptyset) \vdash \{ P \wedge I \} \text{let} m \text{ be } B \text{ in } C \{ P' \wedge I \} [\pi] \\
&\Delta(\emptyset) \vdash \{ P \wedge I \} \text{let } m \text{ be } B \text{ in } C \{ P' \wedge I \} [\pi]
\end{align*}
\]

Figure 7. Selected correctness rules and axioms for commands.

Separators. Given effect sets \( \delta \) and \( \pi \), the separator formula \( \delta \times \pi \) is defined to be the conjunction of certain disjointness formulas. For example, \( rd \ G_1 f \times wr \ G_2 g \) is defined to be \( G_1 \not\equiv G_2 \) if \( f \equiv g \); otherwise, it is just \( \true \). In a state where \( \delta \times \pi \) holds, nothing that the read effects in \( \delta \) allow to be read can be written according to the write effects \( \pi \). The following result, together with Lemma 2, proves soundness of FRAME.

Lemma 3 (separator agreement) Consider any effect sets \( \delta \) and \( \pi \). Suppose \( \sigma \not\Rightarrow \sigma' \vdash \pi \) and \( \sigma \vdash \delta \times \pi \). Then \( \text{Agree}(\sigma, \sigma', \delta) \).

Proof rules and correctness. A correctness judgement takes the form \( \Gamma \vdash \{ P \} C \{ P' \} [\pi] \) and is well formed in \( \Gamma \) just if \( P, P', \pi \) are well formed in \( \Gamma \) and also \( \Gamma \vdash C \). Validity of the judgement consists of a standard part —from any initial state that satisfies \( P \), \( C \) does not fail (terminate with error), and if it terminates then the final state satisfies \( P' \)— and a non-standard one: any allocation and update effects are allowed by \( \pi \) (Def. 1).

Selected proof rules appear in Fig. 7. For field update we choose a “small axiom”, inspired by [28], which snapshots \( F \) with an auxiliary variable \( y \) in the precondition. The effect in the rule is a write to the \( f \) field of the single object in region \( \{ x \} \) that exists in the pre-state. The other rules shown are structural rules. The rule of substitution refers to \( \text{specification-only} \) variables that are not allowed to appear in code.9

4. Region logic with procedures

In this section, procedures are added to the programming language. Correctness judgements are extended with hypotheses, i.e., procedure parameters.

8We do not need to recurse further with \( G_1 \) and \( G_2 \). The \( \text{ftpt} \) function at the core of the frames judgement generates convex sets; if the set contains \( rd \ G_1 f \) then it also contains \( \text{ftpt}(G) \).

9Note that [3] conjectures mistakenly that real effects of commands can be used for the substitution rule; here we give a sound rule.
To variable declarations, declare procedure parameters. Then the update and new commands are allowed in source programs and used only in the semantics to enforce that procedure body where "\( \Delta \)" in commands that may have free occurrences of procedures specified in procedures specified in procedures (4) to be well formed, \( Q, Q', \tau \) should be wf in \( \Gamma \), \( x : T \). Less obviously, \( \tau \) must not contain \( x \); this enforces the usual constraint on value-parameters: so that use of \( x \) in postconditions or in effects like \( \text{wr} x^{f} \) refers to its initial value.

Correctness judgements now take the form

\[ \Delta \vdash^{\tau} \{ P \} C \{ P' \} [\tau] \quad (5) \]

Judgement (5) is well formed just if \( P, P', \tau \), and all specifications in \( \Delta \) are well formed in \( \Gamma \). Moreover \( C \) is well formed with respect to the procedure signatures in \( \Delta \), i.e., \( \Gamma; \text{specs}(\Delta) \vdash C \) where \( \text{specs} \) extracts the procedure signatures from a set of procedure specifications.

There are two plausible interpretations of a hypothetical correctness judgement (5), both expressing that \( C \) is "modularly correct" with respect to the specifications of procedures of its context \( \Delta \). The first interpretation can be described as follows: Consider the semantics of \( C \) started from a procedure environment that links each procedure in \( \Delta \) to an arbitrary command that satisfies the procedure’s specification. That is, \( C \) is correct with respect to all correct implementations of \( \Delta \). The second interpretation avoids this universal quantification by interpreting \( C \) with respect to a single transition relation (not necessarily denotable by a command) that models the least refined or "worst" meaning that satisfies the specification. This embodies all possible behaviors of correct implementations. The second interpretation is popular in work on program refinement. It is used as well by O’Hearn et al [27, 29]. Although typically the second interpretation is associated with a big step semantics, we use it in mixed step form (as in, e.g., [31]).

Fig. 9 completes the definition of transition relation \( \Delta \), by giving the step from configuration \( \langle m(z) \rangle, \sigma, \mu \rangle \) where \( m \) is a context procedure rather than being in the environment \( \mu \). To explain the definition, let us first consider a specification \( \{ P \} m(x : T) \{ P' \} [\tau] \). There are two cases. If \( \sigma \) satisfies the pre-
The judgement is \( \text{wf} \) if the conditions for (5) hold and the read effects \( \overline{\delta} \) and \( \overline{\theta} \) are \( \text{wf} \) in \( \Gamma \).

The current command in a configuration can always be written as a sequence of one or more commands that are not themselves sequences; the first is the active command, the one that is rewritten in the next step. Formally we define \( \text{active}(C_1; C_2) = \text{active}(C_1) \) and \( \text{active}(C) = C \) if there are no \( C_1, C_2 \) such that \( C = C_1; C_2 \).

**Definition 4** A correctness judgement is valid, written

\[
\Delta(\overline{\delta}); \Theta(\overline{\theta}) \vdash^F \{ P \} \ C \ { P' } \ \mathbf{[\varepsilon]} \]

iff the following holds. Let \( \Delta' \) be the union (\( \Delta, \Theta \)), let \( C_0 \) be \( C \), and let \( \mu_0 \) be an arbitrary procedure environment disjoint from the procedures linked within \( C \) or present in \( \Delta, \Theta \). Then for all \( \Gamma \)-states \( \sigma_0 \) such that \( \sigma_0 \vdash P \)

- It is not the case that \( \langle C_0, \sigma_0, \mu_0 \rangle \vdash A_0, \ast; \text{fault} \).
- For all computations \( \langle C_0, \sigma_0, \mu_0 \rangle \vdash A_0, \ast (\text{skip}, \sigma_n, \mu_n) \) of \( n \) steps (noting that \( n \geq 0 \) and \( \mu_n = \mu_0 \)) we have both \( \sigma_0 \vdash P' \) and \( \sigma_0 \sim \sigma_n \vdash \varepsilon \).

Note that in the case \( \overline{\delta} \) and \( \overline{\theta} \) are empty, this coincides with the definition of validity in Sec. 4.

The axioms for assignment forms are now given hypotheses \( \Delta(\overline{\delta}); \Theta(\overline{\theta}) \) with empty dynamic bounds. Proper rules are revised to have \( \Delta(\overline{\delta}); \Theta(\overline{\theta}) \) in both antecedents and conclusions, except that \( \text{CALL} \) and \( \text{LINK} \) are replaced by the ones in Fig. 11. Note that \( \text{LINK}' \) requires the procedure body to respect the other dynamic bounds and linked procedures must have empty dynamic bound. In addition, the Figure adds rules used to introduce and eliminate the dynamic boundary.

**Rule SOF** uses an operation \( \odot I \) that conjoins a formula \( I \) to pre- and post-conditions of specifications. It is defined by

\[
\langle Q \rangle m(x: T)\{ \langle Q' \rangle \} [\varepsilon] \odot I = \langle Q \land I \rangle m(x: T)\{ \langle Q' \land I \rangle \} [\varepsilon]
\]

Define \( \Delta \odot I \) by distributing \( \odot I \) over the specifications in \( \Delta \). Rule SOF also imposes a mild admissibility condition on \( I \). The problem is that some useful invariants include alloc in their effect, e.g., if in the example of Sec. 2 we drop variable \( \text{pool} \) and instead let \( I \) quantify over all allocated \text{Sets}. Typical clients do allocation, and thus write alloc, which would conflict with a dynamic boundary containing \text{rd alloc}. The solutions are based on the limited way that alloc gets written: only by the new command and only by adding the newly allocated object.

Our first solution to the above problem is similar to one used in some specific invariant disciplines [26, 32].

**Definition 5** Formula \( Q \) is admissible with respect to the procedure specifications \( \Delta \), written \( \text{admiss}(Q, \Delta) \), iff it is not falsifiable by allocation. That is, for any \( \sigma \) with \( \sigma \vdash Q \), we also have \( \sigma' \vdash Q \) where \( \sigma' \) is \( \text{new}(\sigma, o, K, \text{default}(\mathcal{T})) \) and \( \mathcal{T} \) is the field types for \( K \) and \( o \) is any fresh reference (not in \( \text{alloc}(\sigma) \)).
It posits a condition $Ok$ is that it makes this rule sound, i.e., it ensures that every step of $C$, other than calls to methods in $\Theta$, respects the bound $\overline{\theta}$.

We leave this topic and turn to the main result of the paper.

**Theorem 7** Any judgement $\Delta(\overline{\delta}); \Theta(\overline{\theta}) \vdash \{ P \} C \{ P' \} \mathcal{[} \pi \mathcal{]}$ is derivable, i.e., $\Delta(\overline{\delta}); \Theta(\overline{\theta}) \vdash \{ P \} C \{ P' \} \mathcal{[} \pi \mathcal{]}$.

**Proof:** By induction on the derivation and by cases on the last rule used. Each case simply appeals to Lemma 9.

**Proposition 8** For all $C, C', \sigma, \sigma', \mu, \mu'$, $\Delta, \Delta'$, such that $\Delta$ and $\Delta'$ declare the same methods and active($C$) is not a call to a method in $\Delta$, we have: $(C, \sigma, \mu) \rightarrow_\Delta (C', \sigma', \mu')$ if and only if $(C, \sigma, \mu) \rightarrow_\Delta (C', \sigma', \mu')$.

Proof: By cases on the inductive structure of $C$; under the given conditions, only the transition rules in Fig. 8 are at play.

**Lemma 9** (rule soundness) Every axiom is valid. For every proper rule, derives valid conclusions from valid antecedents.

**Proof:** By cases on the axioms and rules. We give the main case, SOF. Assume the antecedent is valid, $\Delta(\overline{\delta}); \Theta(\overline{\theta}) \vdash \{ P \} C \{ P' \} \mathcal{[} \pi \mathcal{]}$. To prove validity of the conclusion, i.e.,

$$\Delta(\overline{\delta}); (\Theta \circ I)(\overline{\theta}) \vdash \{ P \} C \{ P' \} \mathcal{[} \pi \mathcal{]}$$

(7)

let $\Delta_a$ be the hypotheses $(\Delta, \Theta \circ I)$, for the conclusion of the rule, and let $\Delta_a$ be the hypotheses $(\Delta, \Theta)$ for the antecedent. Consider any $\sigma_0$ with $\sigma_0 \vdash P \& I$, let $\mu_0$ be any environment disjoint from $\Delta, \Theta$, and let $C_0$ be $C$.

**Claim A:** Consider any computation from $(C_0, \sigma_0, \mu_0)$ under transition relation $\rightarrow_\Delta$. Then (a) that sequence of configurations is also a computation of $\rightarrow_\Delta$ and (b) if $\sigma_0 \vdash I$ then $I$ holds in every configuration. Claim A implies (7) as follows. It is not the case that $(C_0, \sigma_0, \mu_0) \rightarrow_\Delta *$ fault, because that would imply $(C_0, \sigma_0, \mu_0) \rightarrow_\Delta *$ fault which contradicts validity of the antecedent. Furthermore, by validity of the antecedent we get $\sigma_0 \vdash P'$ and $\sigma_0 \vdash \pi$ from the claim. What remains is to show that for all $0 < i \leq n$, either active$(C_{i-1})$ is a call to some method $m$ in $(\Theta \circ I)$ or else active$(\Theta(m, \mu_0, \sigma_0, \overline{\theta}))$, and mutatis mutandis for $\Delta$ and $\overline{\delta}$. This follows immediately from the corresponding condition in validity of the antecedent.

It remains to prove Claim A, which is by induction on the length of the computation sequence for $(C_0, \sigma_0, \mu_0) \rightarrow_\Delta * (C_n, \sigma_n, \mu_n)$. The induction step uses Claim B: For any $D, D', \sigma, \sigma', \mu, \mu'$, suppose that $(D, \sigma, \mu)$ is reachable under $\rightarrow_\Delta$ from an initial configuration $(C_0, \sigma_0, \mu_0)$. Suppose $\sigma \vdash I$ and $(D, \sigma, \mu) \rightarrow_\Delta (D', \sigma', \mu')$. Then $(D, \sigma, \mu) \rightarrow_\Delta (D', \sigma', \mu')$ and $\sigma' \vdash I$. To prove Claim B there are 3 cases: (a) If active$(D)$ is not a call to a context method (noting that $\Delta_a$ and $\Delta_a$ declare the same methods) then by Proposition 8, $(D, \sigma, \mu) \rightarrow_\Delta (D', \sigma', \mu')$. If $\sigma' \neq \text{new}(\sigma, o, K, \overline{\theta})$ then $\Delta_a$ alloc$(\sigma')$ from $\Delta_a$ and $\Delta_a$ (by valid antecedent) and $\sigma \vdash I$, we get $\sigma' \vdash I$ by Lemma 2. In case $\sigma' \neq \text{new}(\sigma, o, K, \overline{\theta})$, $\sigma_0 \vdash I$ and side conditions $I \vdash \text{rd alloc}, \overline{\theta}$ from $\text{Iadmis}(I, \overline{\theta})$ of SOF, we get $\sigma' \vdash I$ by Lemma 6.

For the second and third cases, suppose that active$(D)$ is a call to a method $m$ with specification $(V \{ m(z: T) \} \{ V' \} \{ \pi \})$.

Case (b) $m \in \Delta$: It must be that $\sigma \in \{ V \}$. For, if not, then by context call semantics (Fig. 9) we would have $(D, \sigma, \mu) \rightarrow_\Delta *$ fault.
which, because \(\langle D, \sigma, \mu \rangle\) is reachable, contradicts the antecedent \(\Delta(\theta); \Theta(\bar{\theta}) \vdash^{I} \{ P \} C \{ P' \} \bar{\theta}\). So by the first rule in Fig. 9 for both \(\Delta_{x}\) and \(\Delta_{z}\), we get \(\langle D, \sigma, \mu \rangle \searrow_{\bar{\theta}}^{\Delta_{x}} \langle D', \sigma', \mu' \rangle\). We show that \(I\) is maintained, by cases on whether \(\sigma' = \text{new}(\sigma, \alpha, \bar{K}, \theta)\). If so, from \(\sigma \models I\) and side conditions \(I \vdash_{\text{rd alloc, \bar{\theta}}} \text{from } I\) and \(\text{admiss}(I, \theta)\), we get \(\sigma' \models I\) by Lemma 6. Otherwise, from \(\text{Agree}(\sigma, \sigma', \bar{\theta})\) and \(\sigma \models I\), we have \(\sigma' \models I\) by Lemma 2.

Case (c) \(m \in \Theta \otimes I\). Then \(V = Q \otimes I\) and \(V' = Q' \otimes I\) for some \(Q, Q'\) such that \(\langle Q(m(x); T) \{ Q' \} \bar{\theta} \rangle\) is in \(\Theta\). It must be that \(\sigma \models Q\); for, if not, then by method call semantics (Fig. 9) we would have \(\langle D, \sigma, \mu \rangle \searrow_{\bar{\theta}}^{\Delta} \text{fault, contradicting } \Delta(\theta); \Theta(\bar{\theta}) \vdash^{I} \{ P \} C \{ P' \} \bar{\theta}\). Because \(\sigma \models Q\), by specification of \(m\) in \(\Theta \otimes I\) we get \(\sigma' \models Q' \land I\).\(^{11}\)

\(^{11}\)In part (c) of the proof of the Claim B, the argument would also apply to a \(\sigma'\) arising as a result of a call to a constructor method. The postcondition of such a method must be of the form \(Q' \otimes I\) and thus \(I\) must be established by \(\sigma'\) (replacing the appeal to Lemma 6).

\section*{6. Examples}

**Toy memory manager.** This example is adapted from the one of O’Hearn et al. [27] who use it to exemplify ownership transfer. It also exemplifies situations where clients retain access to locations on which they are not currently allowed to act. Those references must be known when reasoning about the client. This might be done by tracking a “typestate” that distinguishes between, e.g., updatable and non-updatable phases of an object’s life [5]. We will do it by tracking the set of objects that are currently in the “freed” state. This is a “static” module, i.e., unlike our other examples it is not individual instances of a class that represent an abstraction, but rather some global variables.

**global-vars result : Node; freed : rng; module-vars flist : Node; count : int; alloc()**

\[
\begin{align*}
\text{if } count = 0 & \text{ then result : } = \text{ new Node();} \\
\text{else freed : } = \text{ freed } \{\text{ flist;}
\text{ result : } = \text{ flist; flist : } = \text{ flist.nxt; count } = \text{ count } \downarrow ;
\text{ freed : } = \text{ freed } \cup \{n\};
\end{align*}
\]

The interface specifications are in Fig. 12. The module invariant is \(I_{M}\) defined as FC(flist, freed, count). Here FC is defined by induction on the size of its region parameter:

\[FC(f : Node; r : rng, c : int) : (f = null \Rightarrow r = \varnothing \land c = 0) \land (f \neq null \Rightarrow f \in r \land c > 0 \land FC(f.nxt, r - \{f; c - 1\})\]

So \(I_{M}\) says freed is the nodes reached from flist and count is the size. As dynamic boundary we choose \(\Delta_{M} : rd \text{ freed, freed.nxt}\)

To be precise, the read footprint of \(I_{M}\) includes also flist and count, but we shall assume they are hidden from the client by ordinary poking.

The implementation of alloc relies on accuracy of count. In particular, it relies on \(count \neq 0 \Rightarrow \text{flist } \neq \varnothing\) (as otherwise the dereference \(f.nxt\) could fault), but for this to hold on subsequent calls the stronger condition \(I_{M}\) needs to be maintained as invariant.

Consider this strange client that both reads and writes data in the free list — but not in a way that interferes with the module.

\[
\begin{align*}
\text{var } x, y : Node; \quad x : = \text{ new Node();} \\
\text{alloc(); } y : = \text{ result;}
\text{if } x \neq \text{ freed } \land y \neq \text{ freed } \quad \text{ by spec of alloc}
\text{free(x); free(y);}
\text{while } y \neq \text{ null } \{ y.val : = 7; \quad y : = y.nxt; \}
\{ x \in \text{ freed } \land y \in \text{ freed}\}
\end{align*}
\]

This is provable using the specifications (Fig. 12) and rules DYNBNDDINTRO and DYNBNDDINTROM. The point is that although the loop updates val, that effect is separate from the dynamic boundary, \(\Delta_{M}\).

**Combining modules.** Consider a client program that uses both the memory manager and the Set module of Sec. 2:

\[
\text{let alloc, free be } \ldots \text{ in let add, contains, remove be } \ldots \text{ in Cli}
\]
(elidding the parameter lists, implementations, and constructor to save space). Let us write $\Delta$ for the two specifications in Fig. 12 and $\Theta$ for the four in Fig. 2. The main program $Cli$ is verified in context $\Delta(\delta_M); \Theta(\Theta_f)$, i.e., it must respect the dynamic boundary $\delta_M$ for the memory manager and $\Theta_f$ for the Set module. The implementations of $\text{add}$, $\text{contains}$, etc. are verified in context $\Delta(\delta_M); \Theta()$, i.e., they may use the memory manager but must respect its dynamic boundary. Finally, $\text{alloc}$ and $\text{free}$ are verified in context $\Delta()$.

**Observer pattern.** This example illustrates the verification of a client of the Observer pattern (Figs. 13, 14). A subject has a list of observers cognizant of its internal state (here represented as an integer value held in field $\text{val}$ of a subject). A new observer registers itself with a subject (the observer’s $\text{sub}$ field tracks its subject) and is notified of its subject’s state which it caches in field $\text{cacheVal}$. When a subject’s state is updated it notifies each of its observers. Each observer then calls back into the subject interface and gets the new value.

A subject and its observers together form a cooperating cluster of objects that are not in an ownership relation. Accordingly we can consider a module containing both classes $\text{Subject}$ and $\text{Observer}$ whose interface exports the methods $\text{update}$, $\text{get}$ and the constructor for $\text{Observer}$. In Fig. 13, $\text{s.}O$ is a region containing the observers of subject $s$. The subject employs a list data structure to manage its observers — this is hidden from clients as is the list header, $\text{s.}O$. Let $\text{SubH}(s)$ be the predicate $\text{List}(\text{s.obs}, \text{s.O})$ that says “for subject $s$, list beginning at $\text{s.obs}$ lies in region $\text{s.O}$”. The exact definition of $\text{List}$ is immaterial here (but see [3]); it suffices to know that the read effects of $\text{SubH}(s)$ are $\text{rd s}, \{s\}^*\text{obs (O)}, \{s\}^*\text{O.nxt}$. $\text{get}$.

Hidden invariant, $I$, is defined as $I(\Theta, \Theta)$ where $I(m, n)$ is

$(\forall s : \text{Subject} \in \text{alloc} = m \mid \text{SubH}(s)) \land$

$(\forall o : \text{Observer} \in \text{alloc} = n \mid o.\text{sub} \neq \text{null} \Rightarrow o \in o.\text{sub}\{O\})$

The second conjunct says that any observer, $o$, with non-null subject $o.\text{sub}$, is in the subject’s region $O$. The specifications use the predicate $\text{Obs}(b, s, v) : b.\text{sub} = s \land b.\text{cacheVal} = v$. Its read effect is $\text{rd b, s, v} \{b.\text{cacheVal}\}$. Suppose there are two subjects $s, t$ with $s.\text{val} = 0$ and $t.\text{val} = 5$. Here is a client, $C$:

$$o := \text{new Observer}(s); p := \text{new Observer}(t); s.\text{update}(2);$$

Then we can show, using only the specifications in Fig. 14 that after the call to $\text{update}$, $p.\text{cacheVal} = 5$ while $o.\text{cacheVal} = 2$. Let $P' : \forall b : \text{Observer} \in \{s\}^*\text{O} \mid \text{Obs}(b, s, 0)$ $P' : \forall b : \text{Observer} \in \{s\}^*\text{O} \mid \text{Obs}(b, s, 2)$ $Q : \forall b : \text{Observer} \in \{t\}^*\text{O} \mid \text{Obs}(b, t, 5)$ $W : P \land Q \land s.\text{val} = 0 \land t.\text{val} = 5$

in the proof outline.

**Figure 12. Interface specifications for memory manager. Dynamic boundary: rd freed, freed*\text{nxt}**

**Figure 13. Subject/Observer implementation.**

$\{W\}$ $\{P \land Q \land s \neq t \ldots\} \quad \text{by CONSEQ}$

$$o := \text{new Observer}(s);$$

$$\{P \land Q \land o \in \{s\}^*\text{O}\} \quad \text{by s \neq t, spec of Observer, FRAME}$$

$$p := \text{new Observer}(t);$$

$$\{P \land Q \land o \in \{s\}^*\text{O} \land p \in \{t\}^*\text{O}\ldots\} \quad \text{ditto}$$

$s.\text{update}(2);$$

$$\{P' \land Q \land o \in \{s\}^*\text{O} \land p \in \{t\}^*\text{O}\ldots\} \quad \text{by s \neq t, update, FRAME}$$

$$\{o.\text{cacheVal} = 2 \land p.\text{cacheVal} = 5\ldots\} \quad \text{by CONSEQ}$$

Verification of $\text{Cli}$: let $get() \ldots$

$\text{update(n; int) be ...}$

$\text{Observer(u; Subject) be ... in C}$

requires showing (by $\text{LINK}'$) $\vdash \{W \land I\} \quad \text{Cli} \quad \{W' \land I\} \quad \text{sr}$

where $W'$ is $s.\text{val} = 2 \land t.\text{val} = 5$ and $\text{sr}$ is $w_0, p, \text{alloc}$,

$\{\{o\}^*\text{O} \cup \{t\}^*\text{cacheVal}, \{t\}^*\text{O}, t.\text{O.nxt}\}$. Dynamic boundary $\Theta$ is $\text{rd alloc}(O, dg)$, alloc$^*\text{O.nxt}$. It is easy to see $I \vdash (\text{br, rd alloc}) \text{frm } I$. Read effect alloc$^*\text{sub}\{O\}$ of $I$ is subsumed by alloc$^*\text{O}$ because alloc$^*\text{sub} \subseteq \text{alloc}$. We first need

$$\Theta(\Theta_f) \vdash \{W\} \quad C \quad \{W'\} \quad \text{sr}\quad (8)$$

where $\Theta$ is a list containing the specifications in Fig. 14. But using the proof outline above established $\Theta(\Theta_f) \vdash \{W\} \quad C \quad \{W'\} \quad \text{sr}$

from which (8) follows by a few applications of $\text{DYNBNDINTROM}$ and sequencing. Therefore, first by $\text{SOF}$, we can conjoin $I$ and then by $\text{DYNBNDELIM}$ drop $\Theta$ to obtain

$$(\Theta \land I)() \vdash \{W \land I\} \quad \text{C} \quad \{W' \land I\} \quad \text{sr}\quad$$

Now for each $\{Q\}m\{Q'\}n\text{[m]}$ in $\Theta$ we must show

$$(\Theta \land I)() \vdash \{Q \land I\} \quad \text{B}_m \quad \{Q' \land I\} \quad \text{sr}$$

where $Q$ and $Q'$ are the pre- and post-conditions and $\text{sr}$ are the effects according to Fig. 14 and $\text{B}_m$ is the implementation of $m$. For example, to verify Observer’s body, $B_0$, we factor $I$ into $J \land I(\{u\}, \{\text{self}\})$ and establish $\vdash \{Q \land J\} \quad \text{B}_0 \quad \{Q' \land J\} \quad \text{sr}$, where $J$ is $\text{SubH}(u) \land \text{sub} \neq \text{null} \Rightarrow \text{self} \subseteq \text{sub}\{O\}$. Now FRAME yields $\vdash \{Q \land J\} \quad \text{B}_0 \quad \{Q' \land J\} \quad \text{sr}.$

**7. Related work**

The influence of SL on our work is clear. We follow the direction of several recent works [10, 16] that have exploited ghost state
to abstract away low-level reasoning (e.g. maintain reachability information as needed while not saying so explicitly in specs.) The price is that programs need to be instrumented with ghost updates and our specifications are less compact in contrast to [27].

Both our Frame rule and our SOF rule use ordinary conjunction to introduce an invariant, together with side conditions that designate a footprint with respect to which the invariant is separated from the write effect of a command. By contrast, in SL these rules use the separating conjunction $\ast$ which expresses the existence of such a footprint. Reynolds gave a derivation using the rule of conjunction that shows the SOF is not sound in SL (the “conundrum”) such a footprint. In an unpublished proof of admissibility of SOF in region logic, Naumann proved the case of the conjunction rule using that $(I \land \_)$ distributes over $\land$. The admissibility proof essentially says that a use of SOF can be replaced by explicitly conjoining the invariant throughout the proof, justified by many uses of ordinary Frame to introduce not only the invariant but also a chosen footprint.

In earlier work [1] we automatically verified the Observer pattern in Boogie, with the invariant hidden — by postulate. Our current work validates the postulate by way of the SOF. Through Peter Müller (personal communication) we have learnt that our Boogie encodings of regions have been adapted to successfully verify the priority inheritance protocol in real time OS code.

Drossopoulou et al. [13] introduce a general framework to describe verification techniques for invariants based on visible state semantics which requires all invariants to hold on all method call/return boundaries. The framework handles subtyping as well. A general result shows that certain constraints on the framework parameters are sufficient for soundness. A number of ownership type disciplines from the literature are studied as instances of the framework, which promises great improvement in comparing and assessing disciplines. In its present form, the framework does not encompass disciplines like Boogie [4] that rely on state-based encapsulation rather than types or disciplines that deal with design patterns. However, the framework does handle callbacks.

One reaction to the difficulty of hiding invariants is to abandon hiding entirely, in favor of abstraction as advocated by Bierman and Parkinson [6]. They use second order separation logic; method specifications refer to “abstract predicates” that are existentially quantified over the specification. This has been implemented in the jStar prototype [12] based on symbolic execution of SL assertions. Parkinson has reacted to the challenge of invariants over object clusters by advocating explicit but abstract invariants in method specifications [30]. We think more experience with realistic clients is needed to evaluate the practicality of explicitly carrying around a conjunct for the invariant of each abstraction in use.

Smans et al. [34] build a prototype verifier to explore ways to do dynamic framing in the setting of ordinary assertion languages (and also SL), gaining precision through use of location sets and abstraction via pure method calls in assertions. Invariants are addressed via abstraction (model fields and pure methods) also in their recent work [33]. Leino’s Dafny language features dynamic frames of the form $G^\ast$any as well as pure functions that can compute recursive predicates like our FC and List examples [22].

8. Conclusion

We presented an imperative notion of module interface, which complements static scoping constructs with state dependent expression of the dynamic part of the encapsulation boundary. Each primitive action of the client must respect this boundary —it is not enough that the end to end effect of the client respects it. This entails a small step interpretation of specifications. (It is achieved in SL using a big step interpretation but a stronger separation property.) What is achieved is sound and flexible modular reasoning that encompasses various design patterns. We conclude that Hoare’s “mis”match is rather an intricate match between modules and their clients and we show how to get it right.

Apart from the specifications for interface methods, a specifier needs to make explicit the dynamic encapsulation boundary. Then the verifier can fulfill its usual obligations, namely, generate verification conditions (a) for the client by using the interface method specifications and (b) for the bodies of interface methods taking the module invariant into account. But the verifier also needs to verify that each primitive action of the client respects the boundary.

In future work we plan to extend our results to subclasing and inheritance as well as state based representation independence [2]. A particularly exciting challenge is callbacks. Our rendition of the Observer pattern (following Parkinson [30]) has callbacks within the module. It appears that the small-step correctness property enables a SOF rule that supports callbacks back and forth across dynamic encapsulation boundaries.

References


<table>
<thead>
<tr>
<th>Method</th>
<th>Pre-condition</th>
<th>Post-condition</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>update(u)</td>
<td>$\forall b, self, self.val : Observer \in {self}^*O \mid Obs(b, self, self.val)$</td>
<td>$\forall b, Observer \in {self}^*O \mid Obs(b, self, n)$</td>
<td>wr ${self}^*val$, self.O$^*cacheVal$</td>
</tr>
<tr>
<td>get Observer(u)</td>
<td>$\forall b : Observer \in {u}^*O \mid Obs(b, u, u.val)$</td>
<td>$\forall b : Observer \in {u}^*O \mid Obs(b, u, u.val)$</td>
<td>wr ${u}^<em>O^{nxt}$, ${u}^</em>(O, dg)$</td>
</tr>
</tbody>
</table>


