Boogie Meets Regions: a Verification Experience Report

Anindya Banerjee[*1], Mike Barnett[2], and David A. Naumann[**3]

[1] Kansas State University, Manhattan KS 66506 USA
[2] Microsoft Research, Redmond WA 98052 USA
[3] Stevens Institute of Technology, Hoboken NJ 07030 USA

Abstract. We use region logic specifications to verify several programs exhbiting the classic hard problem for object-oriented systems: the framing of heap updates. We use BoogiePL and its associated SMT solver, Z3, to prove both implementations and client code.

1 Introduction

Many programs use dynamically allocated mutable storage, which poses challenges for encapsulation and exacerbates the frame problem: how to specify that "everything else is unchanged" for mutable state not directly named by program variables? Several lines of work approach this problem in terms of *footprints*, i.e., the sets of locations that are written or read by a phrase of a program or specification.

- In the Boogie methodology [5], "owned state" is represented by a mutable ghost field that points to an object's owner if it has one. An *effect specification* ("modifies" clause) that licenses update of an object o implicitly licenses update of the objects transitively owned by o.
- In Separation Logic [21], if P is the precondition of a procedure then the procedure may be viewed as owning the state on which P depends. It has license to modify that part of the heap but no other.
- Kassios [17] shows how to manipulate footprints explicitly, in ghost variables, rather than implicitly (in formulas) or indirectly (via transitive ownership).

A key point is that if the read footprint of a formula or pure expression is disjoint from the write footprint of a command, then the command preserves the value of the formula or expression.

Kassios works in a higher order logic, which can directly express that a formula depends on certain locations. He shows how reasoning idioms developed in Boogie or separation logic can be elegantly and effectively articulated using explicit footprints. His approach does require a single discipline to be imposed on all parts of all programs. He develops a relational refinement calculus, following Hehner [15].

Smans et al [24] explore Kassios' approach in terms of conventional sequential specifications with distinct first order precondition, postcondition, and effect. They focus on framing of pure methods used in specifications for data abstraction. They report on encouraging case studies using a prototype verifier based on an SMT solver.

Banerjee et al [4] also explore the approach using first-order specifications, focusing on foundational justification in terms of a Hoare logic that uses footprint expressions in the effects clause. Their Region Logic features a frame rule inspired by that of separation logic (which in turn adapts Hoare's rule of Invariance) to encompass interference via mutable heap objects. In region logic, the frame rule involves a static analysis for read effects of formulas. The effect **rd** $G \cdot f$ expresses that field f of some objects in G may be read. A region expression G denotes a set of non-null object references and footprints are of the form $G \cdot f$. This treatment caters for notations like $G \cdot f \subseteq G'$ which says that the f-image of G is a subset of G'.[4] Region logic features the use of region fields and variables to encode dynamic frames whereas Smans *et al.* use pure methods. Banerjee *et al.* claim that: "A benefit of treating regions as ghost state is that it can be done using first-order specification languages based on classical logic with modest use of set theory. Thus it fits with mostly-automated tools based on verification condition generation...."

This paper reports on some case studies to investigate Banerjee *et al.*'s claim. We demonstrate the utility of using region specifications for

- local reasoning about data structures (Sect. 4),
- encapsulation, even in the presence of callbacks (Sect. 5),
- layered abstractions, using the examples from Smans et al. [24] but without the need for conditional effects (Sect. 6).

In addition, our experience supports their suggestion that the region logic provides a neutral framework in which disciplines such as ownership can be, but need not be imposed system-wide.

In ongoing work we are investigating decision procedures for region logic primitives such as $G \cdot f \subseteq G'$. Here, we use the Z3 solver [12] which does not have a decision procedure for set theory. This poses another research question: Is a first order axiomatization of set theory effective for automating verification of programs and specifications that encode footprints in ghost state? Our answer is yes (Sect. 9) and no (Sect. 7).

Rather than implement a verification condition generator from scratch for our assertion language, we use the BoogiePL intermediate language advocated at VSTTE'05 by Barnett et al [6]. The BoogiePL tool [13] generates verification conditions suited for SMT solvers such as Simplify [14], the CVC family [8], and Z3 [12]. These integrate multiple decision procedures with heuristic instantiation of quantifiers driven by pragmas called *triggers*. They have been used with some success in verifiers like ESC/Java and Spec#, where user interaction is limited to the insertion of assert and assume statements and loop invariants. Our experiments consist entirely of manually written encodings in BoogiePL of the example programs and specifications, with Z3 as solver. We are therefore in a position to provide triggers on an ad hoc basis.

---

[4] These features are useful for reasoning about simulations, as in the context of information flow [1, 2].

2

2 Relevant features of BoogiePL

A BoogiePL [13] program is a collection of procedure specifications and implementa-tions, together with global variables, uninterpreted functions, and axioms. The Boogie tool generates verification conditions for the procedure implementations with respect to their specifications. These conditions, together with the axioms, are translated into an input format for SMT solvers, in our case Z3. We do not explain that translation: the verification conditions represent the weakest precondition of the program [7]. In-stead we focus on the features provided in BoogiePL and their use for encoding region specifications and programs. Aside from regions, our encodings are similar to those automatically generated by Spec#, but we omit the treatment of source level data types.

Types. The type system of BoogiePL is coarser than that of a high-level object-oriented language: objects are represented by the type **ref** (of which **null** is a distinguished ele-ment). The other primitive types are **bool**, **int**, and **name**, all subtypes of **any**. We use the type **name** for fields. Another subtype of **any** is a type constructor for maps, using array-style notation. In particular, we encode regions as [**ref**]**bool**, denoting a boolean-valued function on references.

Heaps. The heap is modeled as an infinite two-dimensional array, roughly of type [**ref**, **name**]**any**, mapping pairs of object references and field names to values. Logically, all objects "have" all fields, but of course, each object uses only those fields which are defined in the corresponding high-level program. There is no built-in notion of an object being allocated or not. We model it using an explicit boolean field, *alloc*, and assume that the field is set appropriately in the necessary places (see below).

   We declare a single global variable, "$Heap$". All operations involving the heap are explicit in our code and specifications: What would be written $o.f$ in source code, and in region logic specifications, is here written as $Heap[o, f]$. We write "Hp" for the type of heaps. (It is not actually [**ref**, **name**]**any** but rather a refinement that uses a form of type dependency to distinguish integer fields from reference fields but we elide the details [13].) We write **field** for the type of fields, again hiding the dependency notation for field names.

   Source programs only compute heaps h that are self-contained in the sense that for any allocated object o, and for any field f of o, if f's content is a reference, p, then p is also allocated in h. We use an uninterpreted predicate, *GH*, read "is good heap", and in our initial experiments an axiom was used saying that $GH(h)$ implies self-containment. It was not a defining axiom, as we expected for some purposes one would want additional conditions such as typing. It turned out that our results do not even need self-containment. There is also a predicate *GO* ("is good object") with a defining axiom that says $GO(o, h)$ iff $GH(h)$ and o is non-null and o is allocated (i.e., $h[o, alloc]$ is true).

Procedures and functions. Procedures in BoogiePL model the specification of imper-ative code (with optional implementations). Functions introduce arbitrary axiomatiza-tions and model user-defined methods within specifications. All parameters are explicit, including the receiver of an instance method and the return value.

Procedures have *preconditions* and *postconditions* for encoding the assertions they demand of a caller and the guarantees that a caller can assume, respectively. *Free* preconditions and postconditions are not checked (asserted) but instead assumed. Such assumptions model facts that are guaranteed by the high-level programming language. Free preconditions represent facts that a callee can depend on, while free postconditions facts that a caller can depend on. For example, we use a free precondition that the receiver of an instance method is non-null and allocated and a free postcondition that all allocated objects remain so (predicate $NoDeallocs$). In this paper we elide most of the free conditions.

Implementations are written using the usual high-level control structures of assignment statements, conditional statements, and while loops. Loops are allowed to have loop invariants. There are also assert statements, which introduce a proof obligation, and assume statments which introduce unchecked facts. For example, every assignment to the heap is followed by an assumption $GH(Heap)$ which is justified because the source language would make *GH* an all-states invariant. Finally, the statement **havoc** o loses all information about the value of the variable o.

When calls of (source code) methods are allowed in specifications, those methods must be *pure*, i.e., not modify any pre-existing state. All calls to a pure method, m, in specifications are encoded by an uninterpreted function, $\#m$. The axioms defining $\#m$ are either derived from the specification for m [11] or else directly from the implementation of m [24].

Allocation. We encode the high-level source statement $o := $ **new** $T(\dots)$ as

$$\textbf{havoc } o; \textbf{ assume } \neg Heap[o, alloc]; \textbf{ call } T..ctor(\dots);$$

where $T..ctor$ is the procedure that encodes the appropriate constructor for the type T. For brevity we use the high-level statement in this paper. All constructors have a free postcondition that the object is a good object, hence allocated. That is, there is no explicit state change that updates the $alloc$ field to allocate an object, but the effect of calling the constructor provides the same functionality. Constructors also have a free precondition that the object is allocated. The free precondition that o is allocated does not lead to a contradiction with the assumption $\neg Heap[o, alloc]$ because only *checked* preconditions become proof obligations (assertions) at call sites to the constructor. The result is that code within a constructor can use the object, e.g., as a parameter to a call.

The default write effect of any constructor is that only fields of the object under construction can be updated.

Quantifiers. Quantifiers are written in BoogiePL as $(Q\ v \bullet\ body)$ where Q is the kind of the quantifier, v is a list of bound variables and their types and $body$ is a predicate dependent on the variables in v. For axioms involving quantifiers, immediately following the $\bullet$ and before the quantifier body a *trigger* might appear. Triggers are syntactic patterns delimited by {...} and are used to provide hints to the SMT prover on how to instantiate the quantified variable. Triggers are discussed in more detail in Sect. 7.

Modifies clauses. BoogiePL specifications include a *modifies* clause, but this merely lists variables. Thus every mutator method lists $Heap$, and for brevity in this paper

we elide that. More important are the source level effect specifications. The high level specification of a method might include effect **wr** $\langle o \rangle . f$, for a parameter o and field f (this region logic notation corresponds to just $o.f$ in the modifies clause of Spec# or JML). As usual, this translates to a postcondition of the form

$$\forall p : \mathbf{ref}, g : \mathbf{field} \bullet GO(p, \mathbf{old}(Heap)) \Longrightarrow \\ \mathbf{old}(Heap)[p, g] = Heap[p, g] \vee (p = o \wedge g = f) \vee \ldots \tag{1}$$

where the elided part comes from other effect specifications. There is no restriction on modifying any field of an object that is allocated during the method call.

3 Regions

Recall that regions, i.e., reference sets, are encoded by their characteristic functions of type $[\mathbf{ref}]\mathbf{bool}$. Region logic features effects of the form **wr** $this.O.f$ where O is a region type field and f a fieldname. The effect **wr** $this.O.f$ is translated exactly as Equation 1, but replacing $p = o$ with the test whether p is member of $\mathbf{old}(Heap)[this, O]$. Note that write effects are interpreted in the initial state. In our experiments, the only form of region expression used are field dereferences and variables. Region fields and variables serve only for reasoning, but BoogiePL does not distinguish ghost state from program state. (Ghosts would be marked as such in a source level notation like Spec#.)

Although sets are represented by the interpreted type $[\mathbf{ref}]\mathbf{bool}$, this BoogiePL type comes equipped only with equality test and the select/update operations. Other set theoretic operations and predicates can be given straightforward defining axioms. Here are some sample declarations.

> **function** $SingletonSet(\mathbf{ref})$ **returns** $([\mathbf{ref}]\ \mathbf{bool})$;
> **axiom** $\forall r : \mathbf{ref}, o : \mathbf{ref} \bullet \{SingletonSet(r)[o]\}\ SingletonSet(r)[o] \iff r = o$;

> **function** $DisjointSet([\mathbf{ref}]\ \mathbf{bool}, [\mathbf{ref}]\ \mathbf{bool})$ **returns** $(\mathbf{bool})$;
> **axiom** $\forall a : [\mathbf{ref}]\ \mathbf{bool}, b : [\mathbf{ref}]\ \mathbf{bool} \bullet \{DisjointSet(a, b)\}$
> $DisjointSet(a, b) \iff \forall o : \mathbf{ref} \bullet \{a[o]\}\{b[o]\}\ \neg a[o] \vee \neg b[o]$;

In the $DisjointSet$ axiom, either $a[o]$ or $b[o]$ can be triggers for the inner quantifier. In the sequel, we write $\varnothing$ for $EmptySet$, $\{o\}$ for $SingletonSet(o)$, $A \cup B$ for $UnionSet(A, B)$, $A = B$ for $EqualSet(A, B)$ to save space. To avoid clutter in this paper, we omit triggers from most axioms.

Predicate $GR(G, h)$ is defined to say that every object in region G is a good object in h. As mentioned in Section 1, region logic features predicates involving the f-image of a region. For example, region G is closed with respect to field f (of type **ref**) in heap h provided that for every object o in G, the value of $o.f$ is either **null** or in G:

> **function** $RegionClosed(G : [\mathbf{ref}]\mathbf{bool}, h : Hp, f : \mathbf{field})$ **returns** $(\mathbf{bool})$;
> **axiom** $\forall G : [\mathbf{ref}]\mathbf{bool}, h : Hp, f : \mathbf{field} \bullet$
> $RegionClosed(G, h, f) \iff \forall o : \mathbf{ref} \bullet G[o] \Longrightarrow h[o, f] = \mathbf{null} \vee G[h[o, f]]$

```
procedure ListCopy(root : ref) returns (result : ref)
var newRoot, prev, tmp, oldListPtr : ref; newRg, origRg : [ref]bool
{ assume root = null ∨ (origRg[root] ∧ RegionClosed(origRg, Heap, nxt));
   newRoot := null; tmp := null; oldListPtr := root; newRg := ∅;
   if (oldListPtr ≠ null){
      newRoot := new Node(); newRg := newRg ∪ {newRoot};
      prev := newRoot; oldListPtr := Heap[oldListPtr, nxt];
      while (oldListPtr ≠ null){
         tmp := new Node(); newRg := newRg ∪ {tmp};
         Heap[prev, nxt] := tmp; assume GH(Heap);
         prev := tmp; oldListPtr := Heap[oldListPtr, nxt]; }}
   result := newRoot;
   assert (root = null ∧ result = null) ∨ (root ≠ null ∧ RegionClosed(newRg, Heap, nxt)
      ∧ newRg[result] ∧ fresh(old(Heap), Heap, newRg) ∧ DisjointSet(newRg, origRg)); }
```

Fig. 1. List Copy implementation with embedded specifications (see text).

Region G is *fresh* with respect to an initial heap provided its elements were unallocated:

> **function** $fresh(h : Hp, k : Hp, G : [\textbf{ref}]\textbf{bool})$ **returns** (**bool**);
> **axiom** $\forall h : Hp, k : Hp, G : [\textbf{ref}]\textbf{bool} \bullet$
> $fresh(h, k, G) \iff GH(h) \land GR(G, k) \land AllNewRegion(h, G)$

The predicate $AllNewRegion(h, G)$ asserts that all objects in region G are unallocated in heap h. The predicate $DifferenceIsNew(h, G, G')$ holds provided G' is a newly allocated region with respect to both G and h: in other words if o is any object allocated in h, either o is **null** or o is an element of G or o is not an element of G'.

4 Local reasoning example: List Copy

Fig. 1 shows the body of a $ListCopy$ method that takes a linked list of nodes as input and produces as output a new list which is a copy of the original list. Our goal is to verify that if the original list is non-empty and resides in a region —say $origRg$— then the copied list is also non-empty and resides in a region, say, $newRg$, that is fresh and disjoint from $origRg$. (We stress that we are not verifying full functional correctness, namely, that the new list is indeed a copy of the original.) For this purpose, $origRg$ and $newRg$ would be auxiliary variables, scoped over both the requires and ensures clause. Such variables are supported by JML and Region Logic but not the current version of BoogiePL, so for illustration we make them local variables instead of parameters and use assert and assume statements to encode the specification.

At the beginning of every loop iteration, the variable $oldListPtr$ contains a pointer to the remainder of the original list that is yet to be copied. The variable $prev$ contains a pointer to the current end node of the new list: the new list grows when a new node pointed to by tmp is added at its end. All nodes in the copied list exist in $newRg$. This

```
// Observer: fields sub, nxt : ref, cache : int
// methods: ctor (constructor), notify
procedure notify(this : ref){                    procedure ctor(this : ref, s : ref){
    var tmp : int;                                   Heap[this, sub] := s;
    tmp := call get(Heap[this, sub]);                assume GH(Heap);
    Heap[this, cache] := tmp; assume GH(Heap); }     Heap[this, nxt] := null;
                                                     assume GH(Heap);
                                                     call register(s, this); }


// Subject: fields obs : ref, val : int, O : [ref]bool
// methods: ctor (constructor), register, add, update, get
procedure update(this : ref, n : int){
    var o : ref; var r1 : [ref]bool;
    Heap[this, val] := n; assume GH(Heap); o := Heap[this, obs]; r1 := ∅;
    while (o ≠ null){call notify(o); r1 := r1 ∪ {o}; o := Heap[o, nxt]; }
```

Fig. 2. Subject/Observer implementation excerpts.

region is fresh with respect to the heap at the entry to the method. Freshness of each node in $newRg$ is ensured by the allocation of a new object (Sect. 2). The constructor call for $Node$ ensures that tmp is a good object, i.e., non-null and allocated. Since tmp is assumed to be unallocated before the constructor call, $\{tmp\}$ is a fresh region.

The loop invariant below is obtained from the postconditions by standard heuristics, except for the last conjunct which should be mechanically generated for every loop translated from a high-level source program; ghost variable $loopPreHeap$ is the snapshot of the heap before the loop body is entered.

$$newRg[newRoot] \wedge RegionClosed(newRg, Heap, nxt)$$
$$\wedge\, fresh(\textbf{old}(Heap), Heap, newRg) \wedge NoDeallocs(loopPreHeap, Heap)$$

This suffices to prove $DisjointSet(newRg, origRg)$.

5 Encapsulation example: Subject/Observer

Fig. 2 shows a more involved example, *Subject/Observer*. A subject, s, has an internal state in field val and a pointer to a list of observers with typical element, o. The head of the list is reached via $s.obs$ and other observers in the list are reached following the observers' nxt fields. The entire list of observers resides in a region contained in (ghost) field O of s. The observer o maintains a pointer to its subject in its sub field and $o.cache$ contains o's current view of s's internal state.

Method $register$ adds an object o to the region O of a subject, s, and notifies o of s's current state. When the subject's state is $update$d, it notifies all of its observers. The purpose of method $notify$ is to update an observer's view of its subject's internal state. Note that $notify$ is called on an observer from within the $update$ method and this results in a callback to the $Subject$'s get method via $this.sub.get()$.

For the specifications of the methods of Fig. 2, the predicates $SubObs$, Sub, Obs are used (inspired by Parkinson [22]). The predicate $Sub(s, v)$ says that the current internal

function $List(o \colon \textbf{ref}, h \colon Hp)$ **returns** $([\textbf{ref}]\textbf{bool})$;
axiom $\forall o \colon \textbf{ref}, h \colon Hp \bullet o = \textbf{null} \iff List(o, h) = \varnothing$
axiom $\forall o \colon \textbf{ref}, h \colon Hp, r \colon [\textbf{ref}]\textbf{bool} \bullet o \neq \textbf{null} \implies List(o, h) = \{o\} \cup List(h[o, nxt], h)$
axiom $\forall h \colon Hp, k \colon Hp, o \colon \textbf{ref} \bullet List(o, h) = List(o, k) \Longleftarrow$
$\qquad (\forall p \colon \textbf{ref} \bullet p \neq \textbf{null} \wedge List(o, h)[p] \wedge List(o, k)[p] \implies h[p, nxt] = k[p, nxt])$

Fig. 3. List axioms.

state of subject s is v and all observers of s are in a list which lies in region $s.O$. The predicate $Obs(o, s, v)$ says that o is an observer of subject s and that v is o's view of s's internal state. Finally, $SubObs$ is a predicate for the entire aggregate structure comprising an instance of $Subject$ together with its $Observers$. $SubObs(s, v)$ holds for a subject s with internal state v when $Sub(s, v)$ holds and for each observer o in s's list of observers, $Obs(o, s, v)$ holds.

Unlike Parkinson's formulation in separation logic, our version includes the heap as explicit parameter, but that would be hidden in source level syntax. In Fig. 3 we introduce a function $List$ so that $List(o, h)$ is a set containing all objects reachable from o following nxt. Using $List$, the defining axioms for Sub, Obs and $SubObs$ look as follows:

$$Sub(s, v, h) \iff GO(s, h) \wedge h[s, val] = v \wedge List(h[s, obs], h) = h[s, O]$$
$$Obs(o, s, v, h) \iff GO(s, h) \wedge GO(o, h) \wedge h[o, cache] = v \wedge h[o, sub] = s$$
$$SubObs(s, v, h) \iff$$
$$\qquad Sub(s, v, h) \wedge (\forall o \colon \textbf{ref} \bullet GO(o, h) \wedge h[s, O][o] \implies Obs(o, s, v, h))$$

The specification of $update$, omitting effects **wr** $\langle this \rangle \bullet val$ and **wr** $this.O \bullet cache$, is

$$\textbf{requires } SubObs(this, val, Heap) \wedge GO(this, Heap)$$
$$\textbf{ensures } SubObs(this, n, Heap)$$

The implementation of $update$ uses local variables $o, r1$. As observers in the subject's (i.e., $this$'s) list of observers get notified they are put in region $r1$. This leads to the loop invariant that notified observers are up to date:

$$\forall p \colon \textbf{ref} \bullet GO(p, Heap) \wedge r1[p] \implies Obs(p, this, n, Heap)$$

Another loop invariant says that region $this.O$ comprises $r1$ together with $List(o, Heap)$ which is the region containing objects yet to be notified.

$$List(o, Heap) \cup r1 = Heap[this, O]$$

The verification goes through, provided we include the equality axiom listed third in Fig. 3. It actually follows from the first two axioms using induction but here we work in pure first order logic (in which reachability is not finitely axiomatizable, cf. [9]). Note also the apparently superflous variable r in the second axiom. It is needed for triggering (Sect. 7).

8

We consider a client that creates two Subjects and updates one:

$sub0 := \textbf{new } Subject();\ obs0 := \textbf{new } Observer(sub0);\ obs1 := \textbf{new } Observer(sub0);$
$sub := \textbf{new } Subject();\ obs := \textbf{new } Observer(sub);\ \textbf{call } sub0.update(5);$

We are able to verify the following postcondition for the client:

$$DisjointSet(Heap[sub0, O], Heap[sub, O]) \wedge$$
$$SubObs(sub0, 5, Heap) \wedge SubObs(sub, 0, Heap)$$

These assertions say that region $sub0.O$ is disjoint from $sub.O$ and updating the internal state of $sub0$ has no effect on the internal state of sub. The key links: condition $Obs(o, s, v, h)$ implies that $o.sub = s$ and thus $SubObs(s, v, h)$ implies that every o in $s.O$ has $o.sub = s$. This is much like an encoding of ownership (as pointed out in [4] and used in VCC [23]) and it implies that if $o.sub \neq o'.sub$ then $o.sub.O$ is disjoint from $o'.sub.O$.

BoogiePL is modular in the sense that in verifying this client code, the verifier uses only the specifications of the constructors and other methods. The next section considers modularity in more depth.

6 Abstraction and hiding examples

A flaw of the preceding Subject/Observer specification is that it mixes conditions on the internal data structures (the list) with those of interest to clients (including O which could be a model field). In Sect. 6.1 we factor apart the specification so that one part can be hidden from clients, in the manner of Hoare's treatment of invariants [16]. Hiding in this fashion introduces a potentially unsound mis-match between the specs used to verify invocations of a method and those with respect to which its implementation is verified. Such a mis-match can be justified by encapsulation, one technique for which is ownership: roughly, the invariant depends only on owned objects and clients are prevented from writing them.

A formal treatment of hiding, however, is beyond the scope of this paper. The interested reader is encouraged to consult [4] and [20] which address not only hiding but also encapsulation at the granularity of object clusters, as is needed for the Subject/Observer example. The use of regions for cluster encapsulation is similar to the use of regions for ownership.

In Sect. 6.2 we explore the use of a field to hold owned objects. The alternative to hiding is abstraction: Instead of a mis-match between the client specification of a method and its verification conditions, the idea is for client specifications to mention the invariant, but treated opaquely as a pure method —of which the owned objects are the footprint.

6.1 Hiding

As suggested in [4], suppose we put classes $Subject$ and $Observer$ together in a module, giving method $register$ module scope while the other methods are public. Sect. 5

considered an invariant, $SubObs$, that pertains to a single Subject and its Observers. Let us factor it into the externally visible part, $SubObsX$, and a hidden part, $SubObsH$, used only in verification of the implementations of the $Subject$ and $Observer$ methods.

$$SubX(s, v, h) \iff GO(s, h) \wedge h[s, val] = v$$
$$SubObsX(s, v, h) \iff SubX(s, v, h) \wedge \forall o : \textbf{ref} \in h[s, O] \bullet Obs(o, s, v, h)$$

We verified client code, including the previous example, using $SubX$ and $SubObsX$ in place of Sub and $SubObs$ in the method specifications (excepting method $register$ which would be module scoped and not used by clients). The client verifications were successful —confirming that they rely on the disjointness reasoning focused on the O field and not on properties of the list data structure.

The implementations of the methods of $Subject$ and $Observer$ are verified using specifications that use not only $SubObsX$ but also $SubObsH$ where

$$SubObsH(s, v, h) \iff SubH(s, h) \wedge SubObsX(s, v, h)$$
$$SubH(s, h) \iff List(h[s, obs], h) = h[s, O]$$

These verifications go through; they merely repackage the earlier specifications.

To cater for a second order frame rule to justify hiding, Banerjee et al [4] propose to hide a single "module invariant" that for this example could be the conjunction of $\forall s : \textbf{ref} \bullet SubObsH(s, val)$ and

$$\forall p : \textbf{ref}, s : \textbf{ref} \bullet p \in Heap[s, O] \wedge Heap[p, sub] \neq \textbf{null} \implies Heap[p, sub] = s$$

We added these as pre- and post-condition for each method, which entails disjointness reasoning with respect to arbitrary other Subjects. The verifications succeed.

6.2   Ownership and abstraction

Although the clients considered above are well behaved, the proposed hiding is actually unsound since client code like $s.obs := s.obs.nxt$ could falsify $SubH(s)$. Some form of encapsulation is needed to preclude such clients. We now consider ownership, a popular device for achieving heap encapsulation. However, we do not delve into the justification of invariant hiding. Instead, we consider ownership for the alternative to hiding: abstraction. Abstraction has been used for invariants by Müller [19] (model fields), Parkinson [10] (existentially quantified predicates), and others. Smans *et al.* [24] use a pure boolean method $invariant()$ to abstract the invariant in specifications.

Like model fields, pure methods are also useful for properties other than invariants. We explore pure methods and ownership in an example of Smans et al [24]. A stack is implemented in terms of an $ArrayList$, a dynamically resizable array which implements its functionality with a primitive fixed-size array. The interface for $ArrayList$ allows elements to be added at the end of the array, to retrieve or delete the element stored at a specific index, and to query the current size of the array. The interface for $Stack$ is $push$, pop, $empty$. In addition, $Stack$ provides a pure method, $size$, that returns the current number of elements in the stack. This is used in the postconditions of the stack's constructor, which says that its value is zero, and the method $push$, which says $\#size() = \textbf{old}(\#size()) + 1$.

10

```
procedure switch(this : ref, other : ref)
  free requires GO(this, Heap) ∧ (other = null ∨ GO(other, Heap));
  requires other ≠ null ∧ Inv(Heap, this) ∧ Inv(Heap, other);
  requires DisjointSet(Heap[this, fp], Heap[other, fp]);
  ensures Inv(Heap, this) ∧ Inv(Heap, other);
  ensures DisjointSet(Heap[this, fp], Heap[other, fp]);
  ensures #size(Heap, this) = #size(old(Heap), other);
  ensures #size(Heap, other) = #size(old(Heap), this);
  ensures DifferenceIsNew(old(Heap), old(Heap)[this, fp] ∪ old(Heap)[other, fp],
                          Heap[this, fp] ∪ Heap[other, fp]);
  //write effect: wr fp.any, other.fp.any
  ensures ∀ p : ref, f : field • GO(p, old(Heap)) ⟹
          old(Heap)[p, f] = Heap[p, f] ∨ old(Heap)[this, fp][p] ∨ old(Heap)[other, fp][p];
```

Fig. 4. Specification for Stack switch

Now $size$ is implemented using the internal representation of the stack. This internal representation consists of an owned object, the $ArrayList$, and its representation. Field fp will hold references to these, and method $size$ is specified to have read effect **rd** $this.fp$.any. The write effect of methods $push$ and pop is **wr** $this.fp.f$.

An interesting part of the example is the verification of the method $switch$ that exchanges the representations of two stacks. The specifications for most mutators would allow footprints to grow but only by fresh objects, but that is not the case for $switch$. Their collective footprint, however, need not change at all. We choose a specification (Fig. 4) that allows the collective footprint to grow with fresh objects, to cater for benevolent side effects that the $ArrayList$ might have. Note the free precondition guaranteed by the language semantics: the receiver of an instance method is non-null and allocated, while the parameter that would have been explicit in the source language is guaranteed to be allocated only if it is not null. However, we chose to add the explicit precondition that it be non-null. Requiring the two stacks to not be aliased is not enough: we need that their footprints are disjoint. We found the verifier is able to track this for newly allocated stacks

The postcondition that encodes the write effect just says that all fields in the heap retain their old values unless the field is in an object in the footprint of either of the parameters.

We consider client reasoning using the following code:

```
var s1 : ref, s2 : ref; s1 := new Stack(); s2 := new Stack();
assert #size(Heap, s1) = 0 ∧ #size(Heap, s2) = 0;
call push(s2, 5);
assert #size(Heap, s1) = 0 ∧ #size(Heap, s2) = 1;
```

Consider the first assertion: the second conjunct, that #$size$ is zero for $s2$ is directly from the constructor's postcondition (not shown). But for the first conjunct to hold it must be the case that the execution of the constructor for $s2$ did not affect the state that $s1$ depends on. This is done without revealing any of the details of the stack's

11

```
procedure switch(this : ref, other : ref)
{ var tmp : ref;
   tmp := Heap[this, contents];
   Heap[this, contents] := Heap[other, contents];
   Heap[other, contents] := tmp;
   Heap[this, fp] := {this} ∪ {Heap[this, contents]}
          ∪ Heap[Heap[this, contents], ArrayList.footprint];
   Heap[other, fp] := {other} ∪ {Heap[other, contents]}
          ∪ Heap[Heap[other, contents], ArrayList.footprint];
   assume GH(Heap);
}
```

Fig. 5. Implementation for Stack switch

implementation with the following axiom, which expresses that the effect of *size* is **rd** *this.fp*.any.

$$\forall h, k : Hp, o : \mathbf{ref} \bullet$$
$$h[o, fp] = k[o, fp] \wedge (\forall p; \mathbf{ref}, f : \mathbf{field} \bullet h[o, fp][p] \implies h[p, f] = k[p, f])$$
$$\implies \#size(h, o) = \#size(k, o);$$

(Some *GO* conditions are elided.) Together with the default frame condition for constructors (Section 2), this is sufficient for the prover to be able to retain the knowledge about the size of the stack $s1$. The same holds for the second assert statement: the second conjunct is directly from the postcondition for *push* while the first conjunct relies on knowing that calling *push* on $s2$ cannot change the state that $s1$ depends on.

The above axiom is sound as long as *size* does not depend on the field of any object which is not contained in the stack's footprint. Utility of the axiom depends on how easy it is to determine that two states satisfy the antecedent, which in turn depends on the encapsulation of the representation objects.

7 Experiences with prover

Verification failure might be due to (a) weak program specifications (pre- and postconditions), (b) weak loop invariant, and (c) incompleteness of triggers. In the first two cases one receives limited feedback from counterexamples.

A trigger of a universal quantifier is a set of expressions that determines how the SMT solver instantiates the quantifier. In theory the SMT solver can instantiate a universal quantifier with any ground term whatsoever. In practice it is better to limit the number of instantiations in some way so that the solver considers only a finite set of ground instantiations from its e-graph data structure (which is used to provide matching up to equivalence [12]). For example, consider the second axiom in Fig. 3 with the trigger restored:

$$\forall o : \mathbf{ref}, h : Hp, r : [\mathbf{ref}]\mathbf{bool} \bullet \ \{List(o, h) = r\}$$
$$o \neq \mathbf{null} \implies List(o, h) = \{o\} \cup List(h[o, nxt], h)$$

This trigger instructs Z3 to instantiate the quantifier only with those o, h, r for which there is a term $List(o, h) = r$ in its e-graph.

Choosing a proper trigger is a craft. Had we removed the apparently redundant bound variable r and used $List(o, h)$ as a trigger, a matching loop would have arisen because the quantifier could be instantiated by any of $List(o, h)$, $List(o.nxt, h)$, ... We tried experimenting with the trigger $List(o.nxt, h)$ instead but this still did not suffice to verify the invariant $(List(o, h) \cup r1 = Heap[this, O])$ (see Sect. 5). Finally after much experimentation and with help from more experienced colleagues we arrived at the trigger $List(o, h) = r$: the intuition was that we needed one instantiation of the quantified variable r to be $Heap[this, O]$.

At the end of Sect. 1 we posed the question: Is a first order axiomatization of set theory effective for automating verification of programs and specifications that encode footprints in ghost state? We have to answer no, unless one is an expert at triggering quantifiers or has expert colleagues close by. The need for good triggers prevented us from introducing an abstract data type for regions in our specifications.

It is sobering to find out how easy it is to unintentionally introduce inconsistencies into the axiomatization. Boogie's `smoke` option creates multiple verification conditions, one per program path, then injects the statement **assert false** at the end of each path, converts all existing assert statements into assume statements and passes the result to the theorem prover. If the prover is able to establish the **false** assertion then the facts along that path reveal inconsistencies. Our experience has been that this is the single most valuable option in Boogie.

8 Related work

The VCC project [23] is focused on the verification of low-level systems code written in C. It uses regions (sets) to structure the flat address space that the C memory model provides access to. Regions are disjoint sets of memory locations: pointers in C are modeled as offsets into a region. Footprints (read/write effects) are specified using (pure) functions, as opposed to our use of a designated field.

Smans et al [24] report on extensions of the Spec# tool to support dynamic frames where pure methods are used to represent footprints. In this work (and in [17]), footprints are sets of locations, where a *location* pairs an object reference with a field name. The language includes a notation, $\& e.f$, for the location (o, f) where o is the value of expression e; this is needed not only in specifications but in the ghost assignments that instrument code. This treatment offers more fine-grained expressiveness; in particular, it provides for abstraction over field names. Region logic needs a separate means to abstract over field names (notation "any" or model fields), but its instrumentation code can be written in Java and C# without need for the & operator (using classes like `Collection<Object>` for reference sets). It would be interesting to see whether the static analysis for framing in region logic [4] can be adapted to location sets. Besides checking write effects (using two-state postconditions as in our work), Smans *et al.* [24] check read effects in a small-step way: for each primitive expression that reads, a verification condition is generated that says the state read is within the specified read footprint. Our experiments did not check read effects. Modular reasoning is enforced

in [24] like in our Sect. 6.1: In reasoning about code inside a module, the verifier is given an axiom that connects a pure method with its implementation. In reasoning about code outside a module, the verifier is given axioms about the frame of a pure method as well as its postcondition, but not its implementation.

Another difference between our work and [24] is our use of fields rather than pure methods to represent footprints. Fields are interpreted within the theorem prover so that reasoning is potentially more precise than for pure methods, though sometimes at the cost of more ghost assignments (e.g., Fig. 5). Some of the pure methods in [24] are recursively defined, which requires care [11]. Finally, they use conditional effects, for which we did not find a need.

Related work on ownership is discussed in [4].

9   Conclusion

We used BoogiePL to specify and verify several examples using effect specifications in the style of Region Logic [4], inspired by the dynamic frames of Kassios [17]. The standard theories of the underlying SMT solver, Z3, were augmented with some axioms for set theory. Bertrand Russell famously said the advantages of postulation are those of theft, but we found ample opportunity for honest toil in the instrumentation of axioms with triggers. Our toil would have been far greater without help from Rustan Leino, Peter Müller, and Michał Moskal. Nonetheless, we read our results as confirming the efficacy of effect specifications using ghost fields and auxiliaries holding just sets of references, lightweight machinery indeed.

The BoogiePL tool defines an axiomatic semantics for the BoogiePL programming language, insofar as it generates first order verification conditions, and we augmented that semantics with a few assumptions intended to express additional invariants of source language like Java or C#. We are unaware of any formal connection between BoogiePL and an actual language implementation. In his talk at VSTTE'05, J Moore argued that "we should build a [verification] system in some programming language for which we have a mechanically supported formal semantics and a mechanically supported reasoning engine – and [. . .] be able at least to state within the system what it means for our system to be correct. If you are working on a smaller piece of the problem – if you are building a system whose expressive power and implementation is beyond the scope of your own work – then you should find somebody to deal with the problem you are creating!". One use for a syntax-directed proof system like region logic is as bridge between verification conditions and the underlying semantics. The logic has been proved sound, on paper, with respect to a denotational semantics simplified from a Java/JML model that has been encoded by deep embedding in PVS [18]. Region logic has been extended with a second order frame rule which captures hiding of invariants and has been proved admissible, on paper [20]. Given the promising experiences by ourselves and others with dynamic framing, we see the remaining gaps as well worth bridging.

All example programs have been verified in Boogie version 0.90. Complete listings of these and a number of other verified programs are available online and in the accompanying technical report [3].

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.

2. T. Amtoft, J. Hatcliff, E. Rodriguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow. In *Formal Methods: International Conference of Formal Methods Europe*, volume 5014 of *LNCS*, 2008.

3. A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: a verification experience report (extended version). Technical Report MSR-TR-2008-79, Microsoft Research, 2008.

4. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.

5. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

6. M. Barnett, R. DeLine, B. Jacobs, M. Fähndrich, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

7. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

8. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, pages 515–518, 2004.

9. B. Beckert and K. Trentelman. Second-order principles in specification languages for object-oriented programs. In *LPAR*, pages 154–168, 2005.

10. G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

11. Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.

12. L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, pages 183–198, 2007.

13. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Mar. 2005.

14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

15. E. C. R. Hehner. Predicative programming part I. *Commun. ACM*, 27:134–143, 1984.

16. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

17. I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods*, volume 4085 of *LNCS*, pages 268–283, 2006.

18. G. T. Leavens, D. A. Naumann, and S. Rosenberg. Preliminary definition of core JML. Technical Report CS Report 2006-07, Stevens Institute of Technology, 2006.

19. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

20. D. A. Naumann. An admissible second order frame rule in region logic. Technical Report CS Report 2008-02, Stevens Institute of Technology, 2008.

21. P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

22. M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

23. W. Schulte. Building a verifying compiler for C. Presentation at Dagstuhl Seminar 08061 for Types, Logics and Semantics of State, 2008.

24. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, 2008.