Verifying Object-Oriented Code Using Object Propositions

Ligia Nistor Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{lnistor,aldrich}@cs.cmu.edu

Abstract

The modular verification of object-oriented code is made difficult by the presence of aliasing. If the program that we want to verify contains multiple references pointing to the same object, we need to know what predicates hold of that object at any point in time. Clients may depend on properties of the object, but if there are multiple clients, one client may break the property that others depend on. Knowledge of both aliasing and predicates allows us to verify whether clients and implementations are compliant with usage protocols.

We have developed a modular protocol checking approach, by introducing the novel abstraction *object propositions*, that combines predicates and information about object aliasing. In our methodology, even if shared data is modified, we know that an object invariant specified by a client holds. This allows two references pointing to the same object to have a consistent view of the object. Our object invariant is different than a class invariant such as the ones in ESC/Java, as in our system two objects of the same class are allowed to have different invariants.

Although there are separation logic approaches that can be used to specify similar programs, the specifications are complex and not modular. In separation logic, the specification of a method must describe all the heap cells that the method touches, through reading or writing. The exact data shared between objects will then be exposed. With the help of access permissions, we are able to hide the aliasing information when possible. This is very important for software evolution because local changes to the code in a system should not modify the specification of other parts of the system.

1. Introduction

In this paper, we are proposing a method for modular verification of object-oriented code in the presence of aliasing. The seminal work of (Parnas 1972) describes the importance of modular programming, where the information hiding criteria is used to divide the system into modules. In a world where software systems have to be continually changed in order to satisfy new (client) requirements, the software engineering principle of modular programming becomes crucial: it brings flexibility, by allowing changes to one module without modifying the others. There are great benefits: the development time is shortened because different groups of programmers work on separate modules and the whole system is easier to understand.

The formal verification of modules should ideally follow the same principle: the specification and verification of a method should not depend on another method. Only through local reasoning can the verification of object-oriented code become practical. One of the things that make local reasoning difficult are aliases, i.e. the existence of multiple references to the same object. To reason about aliases, the verification system has to be aware of interactions between all the references to the same object. Moreover, if two objects share data, there are situations when the specifications of the methods using these objects should not reveal the shared data or the internal representation of the objects. When possible, shared data should be hidden, especially when the shared information is likely to change. Unfortunately, existing first-order separation logic techniques do not hide shared data. When using separation logic to write the specification of a method, the entire footprint of that method has to be mentioned. By its very nature, separation logic (Reynolds 2002) will require the specification to reveal which parts of the data are shared, which cells in the heap are owned by two objects. For each method using the objects, a mention of shared data is needed. This gives rise to very unmodular verification techniques. There exist versions of higher-order separation logics that try to account for modular verification of code, but they are more complicated and difficult to automate.

Our novel approach does not suffer from these shortcomings. Through the use of access permissions, we are able to hide the shared data that two objects have in common. We

will know only that the two objects have *shared* permission to that data. Our solution is more modular and similar systems (Bierhoff and Aldrich 2007) have been automated.

2. Overview

Our methodology uses predicates over the fields of an object, a logical framework and access permissions (Bierhoff and Aldrich 2007). Access permissions were originally proposed for typestate checking (DeLine and Fähndrich 2004), but here we use them for full verification. They are useful when there are multiple aliases that reference the same object in the program. With the help of permissions, we can track how a reference is allowed to read and/or modify the referenced object and, in the case there were other references to the object, how those references might access the object. The different kinds of permissions are described in Figure 1.

Access through other permissions	Current permission Read/write access	... Read-only
None	unique	-
Read-only	-	immutable
Read/write	share	-

Figure 1. Access permission taxonomy

Immutable permissions (Boyland 2003) grant read-only access to the referenced object and guarantee that no reference has read/write access to the same object.

Unique permissions (Boyland 2003) grant read/write access and guarantee that no other reference has any access to the object.

Share permissions (DeLine and Fähndrich 2004) are the trickiest: they grant read/write access to the referenced object, but other references also have read/write access to the same object. For share permissions to be usable, they must guarantee that they will never violate the predicate invariant of the object.

Our main technical contribution is a novel abstraction, called *object proposition*, that combines predicates with aliasing information about objects. For eg., to express that the object q in Figure 2 has a unique permission to a queue of integers in range $[0, 10]$, we use the object proposition $\mathsf{unique}(q)\ in\ Range(0, 10)$.

We want our checking approach to be modular and to verify that implementations follow their design intent. To prove that a postcondition, i.e. an abstract proposition defined in terms of fields, of a method holds on exiting the method, we have to keep track of the current values of the fields (including for fields of primitive type int). By doing this, we can more easily deduce what object propositions are true at the end of a method, considering that the precondition was true in the beginning. In the same way as (Fahndrich and DeLine 2002) track a set of capabilities, i.e. at each point in the program the type system maintains the keys of the tracked objects that are currently live, we also have a set of keys. Those variables and fields that have a place in the heap will be present in the set of keys and will point to the current value. Thus, we have two kinds of permissions: abstract, represented by the object propositions, and concrete, represented by the field keys. We make a subtle distiction: if we consider an object o having a field f with current value x and the current permission having read/write access, the set will contain $f \rightarrow x$. On the other hand, if the current permission has read-only access, the set will contain $f \rightharpoonup x$, to mean that the field f cannot be modified.

Let's consider the two linked queues q and r that share a common tail p, in Figure 2. In separation logic, the specification of any method has to describe the entire footprint of the method, i.e. all heap locations that are being touched through reading or writing in the body of the method. That is, the shared data p has to be specified in the specification of all methods accessing any of the objects q and r. Using our object propositions, we have to mention only $\mathsf{share}(q)$ ($\mathsf{share}(r)$ respectively) in the specification of a method accessing q (or r). The fact that p is shared between the two aliases is hidden by the access permission share.

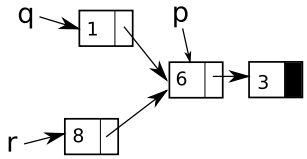

Figure 2. Linked queues sharing the tail

We give the technical details of the differences between verifying this example using separation logic versus object propositions.

3. Example

In Figure 3, we present a simple class and object propositions, that are useful for reasoning about the correctness of client code and about whether the implementation of a method respects the specification. Since they contain access permissions, which represent resources that have to be consumed upon usage, the object propositions are consumed upon usage and their duplication is forbidden. Therefore, we use linear logic (Girard 1987) to write our specifications. Pre- and post-conditions are separated with a linear implication $\multimap$ and use multiplicative conjunction ($\otimes$), additive disjunction ($\oplus$) and existential/universal quantifiers (where there is a need to quantify over the parameters of the predicates).

In our methodology, whenever a new object is created, the object will satisfy the first predicate in the class. At the point where the permission to the object is first split (either into two share permissions, or two immutable ones, see Figure 8), the client can specify the invariant that he expects the object to always satisfy. Different references pointing to the same object, which can be a shared or immutable object, will

```
class Link {
      int val;
      Link next;

  predicate Range(int x, int y) ≡
     val→ v⊗ next→ o
     ⊗ v ≥ x⊗ v ≤ y
     ⊗ [share(o) in Range(x,y) ⊕ o==null]

  predicate NotInRange(int x, int y) ≡
     val→ v ⊗ next→ o ⊗ v < x
     ⊕ v > y
     ⊕ [share(o) in NotInRange(x,y) ⊕ o==null]

  predicate UniRange(int x, int y) ≡
     val→ v⊗ next→ o
     ⊗ v ≥ x ⊗ v ≤ y
     ⊗ [unique(o) in UniRange(x,y) ⊕ o==null]

  void addModulo11(int x)
    share(this) in Range(0,10) —⊸
          share(this) in Range(0,10)
    {val = (val + x)% 11; }

  void addModulo90(int x)
    share(this) in Range(11,100) —⊸
          share(this) in Range(11,100)
    {val = (val + x)% 90 +11; }

  void add(int z)
    ∀ x:int, y:int, x<y .unique(this) in UniRange(x,y)
              —⊸ unique(this) in UniRange(x+z,y+z)
    {val = val + z;
     if (next!=null) {next.add(z);}
    }
}
```

Figure 3. Link class and range predicates

always be able to rely on that invariant when calling methods on the object.

A critical part of our work is allowing clients to depend on a property of a shared object. In this queue example, clients depend on the shared $Link$ items being in a consistent range. Other methodologies such as Boogie (Barnett et al. 2006) allow a client to depend only on properties of objects that it owns and verify object invariants. Our novel verification technique allows a client to depend on properties of objects that it doesn't own too.

To gain read or write access to the fields of an object, we have to $unpack$ it (DeLine and Fähndrich 2004). If we unpack a share or unique permission, we gain modifying access, while immutable permissions only give us read access to the object's fields. After a method finishes working with the fields of a shared object, our proof rules in Figure 7 require that the object be packed to the same predicate as before the unpacking. Since for a unique object, there is only one reference in the system pointing to it and no risk of interferences, we don't require packing to the same predicate for unique objects. We avoid inconsistencies by allowing only one object proposition to be unpacked at the same time. Why? Because otherwise an upacked shared object may not satisfy its invariant any more, but other aliases to it will not know this. Other aliases will rely on the invariant, which is actually broken. It is thus safer to unpack a single object at a time.

The predicate *Range(int x, int y)* in Figure 3 ensures that all the elements in a linked queue starting from the current Link are in the range $[x, y]$. The predicate mentions share(o), thus requiring the existence of at least a share permission (it could be unique) to each Link of the queue. On the contrary, the predicate *UniRange(int x, int y)* requires the presence of a unique permission to all elements in the queue. These restrictions mean that the only method that can be called on a shared Link object satisfying invariant $Range(0, 10)$ is *addModulo11*. The specification of the *addModulo11(int x)* method is the only one that does not break the invariant. Also, if the programmer wants to modify some value in the queue using the *add(int z)* method, the queue must be accessed through a unique permission.

How do we use the predicates and the proof rules in Figure 7 to prove the correctness of code? We use them in the following manner: we try to apply the proof rules from the beginning of a program until the end. Each expression in the program has a designated proof rule in the static semantics and we apply that specific rule. After every line, we write the object propositions that are true at that point. If we can apply all the needed rules from the beginning until the end of the program, it means the program is correct. If not, there must be an error in the program.

When first creating object la in Figure 4, we get a unique permission to it. We then split this permission into two share permissions, as we have to pass a permission to la to the constructor of *Link* when creating p. Since la can be accessed through p from now on, we do not need to mention la in the following object proposition sets. In this respect, our system is an affine one (as opposed to a linear one), because we can drop a resource if we do not need it any more.

When creating the object q, we need to pass in a share permission to p. We obtain it by splitting the current unique permission to p into two share permissions. Before calling method *addModulo11* on r, we also have to split the permission to r so that a share permission can be passed to the method. The splitting of permissions is similar when we call *addModulo11* on q.

The specification in separation logic is more cumbersome and unable to hide shared data. To express the fact that all

```
...
  {}
Link la = new Link(3, null);
  {unique(la) in Range(0,10)}

  {share(la) in Range(0,10) ⊗ share(la) in Range(0,10)}
Link p = new Link(6, la);
  {unique(p) in Range(0,10), share(la) in Range(0,10)}

  {share(p) in Range(0,10) ⊗ share(p) in Range(0,10)}
Link q = new Link(1, p);
  {share(p) in Range(0,10), unique(q) in Range(0,10)}
Link r = new Link(8, p);
  {unique(q) in Range(0,10), unique(r) in Range(0,10)}

  {unique(q) in Range(0,10), share(r) in Range(0,10)
   ⊗ share(r) in Range(0,10)}
r.addModulo11(9);
  {unique(q) in Range(0,10), share(r) in Range(0,10)}

  {share(q) in Range(0,10) ⊗ share(q) in Range(0,10),
  unique(r) in Range(0,10)}
q.addModulo11(7);
  {share(q) in Range(0,10), unique(r) in Range(0,10)}
...
```

Figure 4. Object proposition verification

values in a segment of linked elements are in the interval $[n_1, n_2]$, we need to define the following predicate :

$Listseg(r, p, n_1, n_2) \equiv (r = p) \vee (r \rightarrow (i, s) \star Listseg(s, p, n_1, n_2) \wedge n_1 \leq i \leq n_2)$.

This predicate states that either the segment is null, or the val field of r points to i and the $next$ field points to o, such that $n_1 \leq i \leq n_2$, and the elements on the segment from o to p are in the interval $[n_1, n_2]$. The verification of the same code in separation logic is given in Figure 5.

In separation logic, the natural pre and post-conditions of the method *addModulo11* are $Listseg(this, null, 0, 10)$, i.e., the method takes in a list of elements in $[0, 10]$ and returns a list in the same range.

Thus, before calling `addModulo11` on r, we have to combine $Listseg(r, p, 0, 10) \star Listseg(p, null, 0, 10)$ into $Listseg(r, null, 0, 10)$. We observe the following problem: in order to call `addModulo11` on q, we have to take out $Listseg(p, null, 0, 10)$ and combine it with $Listseg(q, p, 0, 10)$, to obtain $Listseg(q, null, 0, 10)$. But the specification of the method does not allow it, hence the missing step in Figure 5. The specification of `addModulo11` has to be modified instead, by mentioning that there exists some sublist $Listseg(p, null, 0, 10)$ that we pass in, and which gets passed back out again. The modification is unnatural: the specification of `addModulo11` should not care that it receives a list made of two separate sublists, it should only care that it receives a list in range $[0, 10]$.

```
...
  {}
Link la = new Link(3, null);
  {Listseg(la, null, 0, 10)}
Link p = new Link(6, la);
  {Listseg(p, null, 0, 10)}
Link q = new Link(1, p);
  {Listseg(q, p, 0, 10) ⋆ Listseg(p, null, 0, 10)}
Link r = new Link(8, p);
  {Listseg(q, p, 0, 10) ⋆ Listseg(r, p, 0, 10)⋆
   Listseg(p, null, 0, 10)}

  {Listseg(q, p, 0, 10) ⋆ Listseg(r, null, 0, 10)}
r.addModulo11(9);
  {Listseg(q, p, 0, 10) ⋆ Listseg(r, null, 0, 10)}
    *****missing step*****
  {Listseg(q, null, 0, 10) ⋆ Listseg(r, p, 0, 10)}
q.addModule11(7);
  {Listseg(q, null, 0, 10) ⋆ Listseg(r, p, 0, 10)}
...
```

Figure 5. Separation logic verification

This situation is very problematic because the specification of `addModulo11` involving sublists becomes awkward. We can imagine an even more complicated example, where there are three sublists that we need to pass in and out of `addModulo11`. It is impossible to know, at the time when we write the specification of a method, on what kind of shared data that method will be used. Separation logic approaches will thus have a difficult time trying to verify this kind of code, while object propositions allow us to call methods on both lists, no manipulation of predicates being necessary.

4. Grammar

The programming language that we are using is inspired by Feather-weight Java (Igarashi et al. 2001), extended to include object propositions. We have retained only the most important Java concepts for us, that are relevant for showing our methodology. Thus, we have not taken into account features such as inheritance, casting or dynamic dispatch that are too complex and not important for making our point.

Figure 6 shows the syntax of our simple class-based object-oriented language. The proof system composed of the rules for the different expressions in the language and the rules for reasoning about the predicates will be described in the next section.

5. Proof Rules

This section describes the proof rules in Figure 7 used for guaranteeing at compile-time that protocol violations will not occur at runtime. The judgement to check an expression e is of the form $\Gamma; \Pi \vdash_C e : \exists x : T.P$. This is read "in valid context Γ and linear context Π, an expression e

program	PR	::=	$\langle \overline{CL}, e \rangle$
class decls.	CL	::=	$\texttt{class}\ C\ \{\ \overline{F}\ \overline{Q(\overline{x}) = P}\ \overline{M}\}$
field decls	F	::=	$f : T$
methods	M	::=	$T\ m(\overline{T\ x}) : MS = e$
method specs	MS	::=	$P_1 \multimap P_2$
predicates	P	::=	$q \mid x \mid P_1\ \texttt{binop}\ P_2 \mid P_1 \otimes P_2$
		|	$P_1 \oplus P_2 \mid 1 \mid 0 \mid \top \mid f \rightarrow x$
		|	$f \rightharpoonup x \mid \exists z.P \mid \forall z.P$
binary ops	$binop$	::=	$+ \mid - \mid \% \mid =$
		|	$\leq \mid < \mid \geq \mid >$
terms	t	::=	$x \mid o$
		|	$\texttt{true} \mid \texttt{false} \mid t_1\ \texttt{or}\ t_2$
		|	$t_1\ \texttt{and}\ t_2 \mid \texttt{not}\ t$
expressions	e	::=	$t \mid f \mid \texttt{assign}\ f := t$
		|	$\texttt{new}\ C(\overline{t}) \mid t_o.m(\overline{t})$
		|	$\texttt{if}(t, e_1, e_2)$
		|	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$
		|	$t_1\ \texttt{binop}\ t_2$
		|	$\texttt{unpack}\ r\ \ \texttt{from}\ P\ \texttt{in}\ e$
		|	$\texttt{pack}\ r\ \texttt{to}\ P\ \ \texttt{in}\ e$
references	r	::=	$x \mid o$
types	T	::=	$C \mid \texttt{bool} \mid int$
permissions	q	::=	$Perm(\overline{r})\ in\ Q(\overline{r})$
		|	$\texttt{unpacked}(r, q) \mid none(r)$
permission kind	$Perm$	::=	share | immutable | unique

Figure 6. Language and Permission syntax.

executed within receiver class C has type T and satisfies object proposition P". By writing $\exists x$, we bound the variable x to the result of the expression e.

The rule TERM formalizes the standard logical judgement for existential introduction. The notation $[e'/x]e$ substitutes e' for occurences of x in e. The FIELD rule checks field accesses analogously.

NEW checks object construction. The parameters $\overline{t}$ passed to the object constructor have to satisfy initialization predicate Q_0 and become the object's initial field values. The new existentially qualified object is naturally associated with a unique permission to the root state. The $init$ judgement looks up the initialization predicate for a class.

IF is the rule that introduces disjunctive types in the system and checks if-expressions.

LET checks a let binding and makes sure that outdated object propositions are not available after assignments.

The rules for packing and unpacking are PACK-SH-IMM, PACK-UNI, UNPACK-SH-UNI and UNPACK-IMM. When we pack an object to a share permission, we have to pack it to the same object proposition that was true before the object was unpacked. This restriction also holds for immutable permissions because, by definition, the object will not be modified, so it will satisfy the object proposition that was true before unpacking. The restriction is not necessary for unique permissions: objects that are packed to unique permissions can be packed to a different object proposition that the one that was true for them before unpacking. There is a special rule for the unpacking of immutable objects because we want to emphasize that all their fields are also immutable. This restricts the current system by requiring that the predicates of immutable objects should only contain immutable permissions to their fields. Although special predicates have to be written for immutable objects, the soundness of the system is preserved. Since we allow only one unpacked object at any time in the system, we have to keep track if there currently exists an unpacked object. We do this by having a flag ok that shows if there is another unpacked object in the system. If there is, $ok = false$ and we are not allowed to unpack another object. When we apply one of the PACK rules, ok becomes $true$ because the only unpacked object in the system has just been packed.

The CALL rule simply states what is the object proposition that holds about the result of the method being called. This rule first identifies the specification of the method (using the helper judgement MTYPE) and then goes on to state the object proposition holding for the result.

The rule ASSIGN assigns an object t to a field f_i and returns the old field value as an existential x. For this rule to work, the current object $this$ has to be unpacked and to not have permission immutable (because objects with immutable permissions do not allow modification of their fields).

5.1 Permission Splitting

We need a mechanism for tracking permissions in our verification approach.

When a client has a unique permission to a shared object and an alias capturing a shared permission to the object is created, we want the client to still have access to the collection. How can this be achieved? By saying that the client now has a shared permission to the shared object. This mechanism is called permission splitting. Before method calls we split the original permission into two, one of which is retained by the caller. The splitting rules reflect the fact that the permission assumptions are not violated: a `unique` permission is never duplicated and no `immutable` permission co-exists with a `share` permission. The splitting rules are the following.

$\mathsf{unique}(x) \Rightarrow \mathsf{share}(x) \otimes \mathsf{share}(x)$

$\mathsf{share}(x) \Rightarrow \mathsf{share}(x) \otimes \mathsf{share}(x)$

$\mathsf{immutable}(x) \Rightarrow \mathsf{immutable}(x) \otimes \mathsf{immutable}(x)$

This leads to the rules in Figure 8.

The next natural idea is that we would like to reverse permission splits to regain the initial permission to the shared object. This idea is called *permission joining* and can be allowed if we introduce the notion of fractions (Boyland 2003). We will study permission joining in our future work.

$$\frac{\Gamma \vdash t : T \quad \Gamma; \Pi \vdash [t/x]P}{\Gamma; \Pi \vdash t : \exists x : T.P}\ \textsc{Term} \qquad \frac{f_i : T \textit{ is a local field of } C \quad \Gamma; \Pi \vdash [f_i/x]P}{\Gamma; \Pi \vdash f_i : \exists x : T.P}\ \textsc{Field}$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{t : T} \quad \mathsf{init}(C) = \langle \exists \overline{f : T}.Q_0(\overline{f}) \rangle \\ Q_0(\overline{x}) = P \in C \quad \Gamma; \Pi \vdash [\overline{t}/\overline{x}]P\end{array}}{\Gamma; \Pi \vdash \mathtt{new}\ C(\overline{t}) : \exists y : C.\mathsf{unique}(y)\ \mathtt{in}\ Q_0(\overline{t})}\ \textsc{New}$$

$$\frac{\begin{array}{c}\Gamma; (\Pi, t = \mathtt{true}) \vdash e_1 : \exists x : T.P_1 \\ \Gamma \vdash t : \mathtt{bool} \quad \Gamma; (\Pi, t = \mathtt{false}) \vdash e_2 : \exists x : T.P_2\end{array}}{\Gamma; \Pi \vdash \mathtt{if}(t, e_1, e_2) : \exists x : T.P_1 \oplus P_2}\ \textsc{If}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_1 \vdash e_1 : \exists x : T_1.P_1 \\ (\Gamma, x : T_1), (\Pi_2, P_1) \vdash e_2 : \exists y : T_2.P_2\end{array}}{\Gamma; (\Pi_1, \Pi_2) \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \exists y : T_2.[e_1/x].P_2}\ \textsc{Let}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash_C r : T.[\overline{r_1}/\overline{x}]P \otimes \mathsf{unpacked}(r, Perm(r)\ in\ Q(\overline{r_1})) \otimes ok = false \\ \Gamma; (\Pi', Perm(r)\ in\ Q(\overline{r_1}), ok = true) \vdash_C e : E \\ Q(\overline{x}) = P \in C\end{array}}{\Gamma; (\Pi, \Pi', ok = false) \vdash_C \mathtt{pack}\ r\ \mathtt{to}\ Perm(r)\ in\ Q(\overline{r_1})\ \mathtt{in}\ e : E}\ \textsc{Pack-Sh-Imm}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash_C r : T.[\overline{r_2}/\overline{x}]P_2 \otimes \mathsf{unpacked}(r, \mathsf{unique}(r)\ in\ Q_1(\overline{r_1})) \otimes ok = false \\ \Gamma; (\Pi', \mathsf{unique}(r)\ in\ Q_2(\overline{r_2}), ok = true) \vdash_C e : E \\ Q_1(\overline{x}) = P_1 \in C \qquad Q_2(\overline{x}) = P_2 \in C\end{array}}{\Gamma; (\Pi, \Pi', ok = false) \vdash_C \mathtt{pack}\ r\ \mathtt{to}\ \mathsf{unique}(r)\ in\ Q_2(\overline{r_2})\ \mathtt{in}\ e : E}\ \textsc{Pack-Uni}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash_C r : T.Perm(r)\ in\ Q(\overline{r_1}) \\ \Gamma; (\Pi', [\overline{r_1}/\overline{x}]P, ok = false, \mathsf{unpacked}(r, Perm(r)\ in\ Q(\overline{r_1})) \vdash_C e : E \\ Q(\overline{x}) = P \in C \quad Perm \neq \mathsf{immutable}\end{array}}{\Gamma; (\Pi, \Pi', ok = true) \vdash_C \mathtt{unpack}\ r\ \mathtt{from}\ Perm(r)\ in\ Q(\overline{r_1})\ \mathtt{in}\ e : E}\ \textsc{Unpack-Sh-Uni}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash_C r : T.\mathsf{immutable}(r)\ in\ Q(\overline{r_1}) \\ \Gamma; (\Pi', [\overline{r_1}/\overline{x}]P, ok = false, \mathsf{unpacked}(r, \mathsf{immutable}(r)\ in\ Q(\overline{r_1})) \vdash_C e : E \\ Q(\overline{x}) = P \in C \qquad \mathtt{all\ permissions\ present\ in\ P\ must\ be\ immutable}\end{array}}{\Gamma; (\Pi, \Pi', ok = true) \vdash_C \mathtt{unpack}\ r\ \mathtt{from}\ \mathsf{immutable}(r)\ in\ Q(\overline{r_1})\ \mathtt{in}\ e : E}\ \textsc{Unpack-Imm}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : C_0 \quad \Gamma \vdash \overline{t : T} \\ \Gamma; \Pi \vdash [t_0/this][\overline{t}/\overline{x}]P_1 \\ mtype(m, C_0) = \forall \overline{x : T}.\exists result : T_r.P_1' \multimap P_2 \\ P_1\ implies\ P_1'\end{array}}{\Gamma; \Pi \vdash t_0.m(\overline{t}) : \exists\ result : T_r.[t_0/this][\overline{t}/\overline{x}]P_2}\ \textsc{Call}$$

$$\frac{class C\{...\overline{M}...\} \in \overline{CL} \qquad T_r m(\overline{Tx}) : P_1 \multimap P_2 : \exists result : T_r. = e \in \overline{M}}{mtype(m, C) = \forall \overline{x : T}.\exists result : T_r.P_1 \multimap P_2}\ \textsc{Mtype}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash t : \exists x : T_i.Perm_0(t)\ in\ Q_0(\overline{r_0}) \\ \Gamma; \Pi' \vdash f_i : T_i.Perm'(o)\ in\ Q'(\overline{r'}) \otimes p \\ p = \mathsf{unpacked}(this, Perm(this)\ in\ Q(\overline{r})) \\ \Pi' \vdash this.f_i \to o \qquad Perm \neq immutable\end{array}}{\Gamma; (\Pi, \Pi') \vdash (\mathtt{assign}\ f_i := t)\ : T_i.\exists x : Perm'(x)\ in\ Q'(\overline{r'}) \otimes Perm_0(t)\ in\ Q_0(\overline{r_0}) \otimes p \otimes \quad this.f_i \to t}\ \textsc{Assign}$$

Figure 7. Proof Rules

6. Verification of Implementations

Now, we want to verify that a method implementation respects the specification. Whenever we can apply the proof rules we presented in Figure 7, we apply them. But those proof rules do not deal with primitive types. When fields or variables on the heap are assigned primitive types, we will use the standard linear logic and arithmetic rules presented in the Appendix.

$$\frac{}{\mathsf{unique}(r)\ in\ Q(\bar{t}) \vdash \mathsf{share}(r)\ in\ Q(\bar{t}) \otimes \mathsf{share}(r)\ in\ Q(\bar{t})}\ (\textsc{Uni-Sh})$$

$$\frac{}{\mathsf{share}(r)\ in\ Q(\bar{t}) \vdash \mathsf{share}(r)\ in\ Q(\bar{t}) \otimes \mathsf{share}(r)\ in\ Q(\bar{t})}\ (\textsc{Sh-Sh})$$

$$\frac{}{\mathsf{immutable}(r)\ in\ Q(\bar{t}) \vdash \mathsf{immutable}(r)\ in\ Q(\bar{t}) \otimes \mathsf{immutable}(r)\ in\ Q(\bar{t})}\ (\textsc{Imm-Imm})$$

Figure 8. Rules for splitting permissions

We verify that the method $addModulo11()$ from Figure 3 satisfies the implementation. We will assume that $this$ is the caller of the method. Since according to our system any object is packed before calling a method, $this$ has to be first unpacked. When verifying a method implementation, the first step that we must do is to skolemize the method specification if it is universally quantified. The specification in this case is not universally quantified so it will not be skolemized. To be able to formally deduce the postcondition, it is necessary that we unwrap the meaning of the predicate *Range(0,10)*. Also , the rules in Figure 7 do not allow us to simultaneously read and write to a field. Instead, we have to rewrite the assignment $val = (val + x)\%11$ as $let\ t = val\ in\ val = (t + x)\%11$. We obtain the following setting:

$\{val \to x_1 \otimes next \to o_1 \otimes x_1 \geq 0 \otimes x_1 \leq 10 \otimes$
$\mathsf{share}(o_1)\ in\ Range(0,10) \oplus o_1 == null\}$
$let\ t = val\ in$
$\{t \to x_1 \otimes next \to o_1 \otimes x_1 \geq 0 \otimes x_1 \leq 10 \otimes$
$\mathsf{share}(o_1)\ in\ Range(0,10) \oplus o_1 == null\}$
$val = (t + x)\%11$
$\{val \to x_2 \otimes x_2 = (x_1 + x)\%11$
$\otimes\ next \to o_1 \otimes x_2 \geq 0$
$\otimes\ x_2 \leq 10 \otimes \mathsf{share}(o_1)\ in\ Range(0,10) \oplus o_1 == null\}$

For obtaining the object propositions in the last step, we apply the standard linear logic and arithmetic rules from the Appendix. Thus, we are able to prove that the postcondition holds and the implementation of $addModulo11$ satisfies the specification.

7. Specification for Modularity

To illustrate the modularity issues, we present here a more realistic example than the queue example from Section 3. In Figure 9 we depict a simulator for two queues of jobs, containing large jobs (size>10) and small jobs (size<11). The example is relevant in queueing theory, where an optimal scheduling policy might separate the jobs in two queues, according to some criteria. The role of the control is to make each producer/consumer periodically take a step in the simulation. We have modelled two FIFO queues, two producers, two consumers and a control object. Each producer needs a pointer to the end of each queue, for adding a new job, and a pointer to the start of each queue, for initializing the start of the queue in case it becomes empty. Each consumer has a pointer to the start of one queue because it consumes the element that was introduced first in that queue. The control has a pointer to each producer and to each consumer. The queues are shared by the producers and consumers, thus giving rise to a number of aliased objects with *share* permissions.

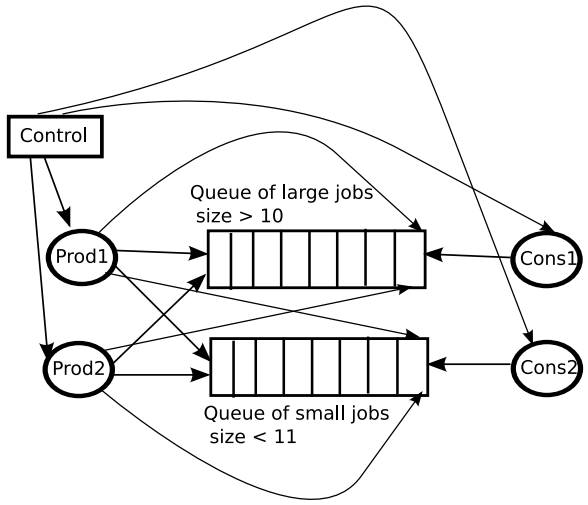

Figure 9. Simulator for queues of jobs

Now, let's say the system has to be modified, by introducing two queues for the small jobs and two queues for the large jobs, see Figure 10. Ideally, the specification of the control object should not change, since the consumers and the producers have the same behavior as before: each producer produces both large and small jobs and each consumer accesses only one kind of job. We will show in the following sections that our methodology does not modify the specification of the control object, thus allowing us to make changes locally without influencing other code, while separation logic approaches (Distefano and Parkinson 2008) will modify the specification of the controller.

The code in Figures 11, 12 and 13 represents the initial running example from Figure 9, that we discussed in the Overview section. The predicates and the specifications of each class explain how the objects and methods should be used and what is their expected behavior. For eg., the Producer object has access to the two queues, it expects the queues to be shared with other objects, but also that the

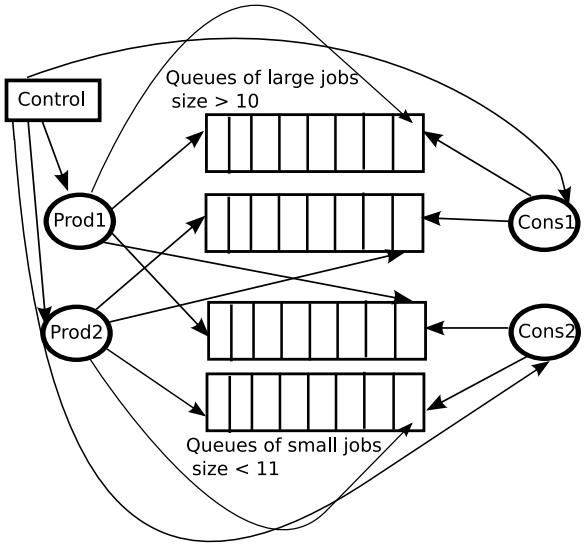

Figure 10. Modification of the simulator

elements of one queue will stay in the range [0,10], while the elements pf the second queue will stay in the range [11,100].

Now, let's imagine changing the code to reflect the modifications in Figure 10. The internal representation of the predicates changes, but their external semantics stays the same: the producers produce jobs and they direct them to the appropriate queue, each consumer accesses only one kind of queue (either the queue of small jobs or the queue of big jobs), and the controller is still the manager of the system. The predicate `BothInRange()` of the Producer class is exactly the same. The predicate `ConsumeInRange(x,y)` of the Consumer class changes to `ConsumeInRange(x,y)` $\equiv$ `startJobs1`$\rightarrow o_1 \otimes$ `startJobs2`$\rightarrow o_2$
$\otimes$ `share(`o_1`) in Range(x,y)`
$\otimes$ `share(`o_2`) in Range(x,y)`.

The predicate `WorkingSystem()` of the Control class does not change.

The local changes did not influence the specification of the Control class, thus conferring greater flexibility and modularity to the code.

The current separation logic approaches do not provide this modularity. Distefano and Parkinson (2008) introduced jStar, an automatic verification tool based on separation logic aiming at programs written in Java. Although they are able to verify various design patterns and they can define abstract predicates that hide the name of the fields, they do not have a way of hiding the aliasing. In all cases, they reveal which references point to the same shared data, and this violates the information hiding principle. By using access permissions, we can hide what is the data that two objects share. We present what are the specifications needed to verify the code in Figure 9 using separation logic.

```
public class Producer {
      Link startSmallJobs, startLargeJobs;
      Link endSmallJobs, endLargeJobs;

  predicate BothInRange() ≡ startSmallJobs→ o1
            ⊗ startLargeJobs→ o2
            ⊗ share(o1) in Range(0,10)
            ⊗ share(o2) in Range(11,100)

  public Producer(Link ss, Link sl, Link es, Link el) {
        startSmallJobs = ss;
        startLargeJobs = sl;
        ...}

  public void produce()
   share(this) in BothInRange() —∘
     share(this) in BothInRange() {
   Random generator = new Random();
   int r = generator.nextInt(100);
   Link l = new Link(r, null);
   if (r <= 10)
   {   if (startSmallJobs == null)
         { startSmallJobs = l;
          endSmallJobs = l;}
    else
       {endSmallJobs.next = l;
       {endSmallJobs= l;}
   }
   else
   {   if (startLargeJobs == null)
         { startLargeJobs = l;
          endLargeJobs = l;}
    else
       {endLargeJobs.next = l;
        endLargeJobs = l;}
   }
  }
}
```

Figure 11. Producer class

The predicate for the Producer class is $Prod(this, ss, es, sl, el)$, where :
$Prod(p, ss, es, sl, el) \equiv p.startSmallJobs \rightarrow ss \star$
$p.endSmallJobs \rightarrow es \star p.startLargeJobs \rightarrow sl \star$
$p.endLargeJobs \rightarrow el$.
The precondition for the `produce()` method is:
$Prod(p, ss, es, sl, el) \star Listseg(ss, null, 0, 10) \star$
$Listseg(sl, null, 0, 10)$.
The predicate for the Consumer class is
$Cons(c, s) \equiv c \rightarrow s$.
The precondition for the `consume()` method is:
$Cons(c, s) \star Listseg(s, null, 0, 10)$.
The predicate for the Control class is :
$Ctrl(ct, p1, p2, c1, c2) \equiv ct.prod1 \rightarrow p1 \star$

```
public class Consumer {
  predicate ConsumeInRange(int x, int y) ≡
          startJobs→ o ⊗ share(o) in Range(x,y)

      Link startJobs;

  public Consumer(Link s) {
        startJobs = s;

  public void consume()
    ∀ x:int, y:int.share(this) in ConsumeInRange(x,y)
        —∘ share(this) in ConsumeInRange(x,y)
    { if (startJobs != null)
       {System.out.println(startJobs.val);
       startJobs = startJobs.next;}
  }
```

Figure 12. Consumer class

$ct.prod2 \rightarrow p2 \star ct.cons1 \rightarrow c1 \star ct.cons2 \rightarrow c2$.

The precondition for `makeActive()` is:

$Ctrl(this, p1, p2, c1, c2) \star Prod(p1, ss, es, sl, el)$
$\star\, Prod(p2, ss, es, sl, el) \star Cons(c1, sl)$
$\star\, Cons(c2, ss) \star Listseg(ss, null, 0, 10)$
$\star\, Listseg(sl, null, 0, 10)$.

The lack of modularity will manifest itself when we add the two queues as in Figure 10.

The predicates $Prod(p, ss, es, sl, el)$ and $Ctrl(ct, p1, p2, c1, c2)$ do not change, while the predicate $Cons(c, s1, s2)$ changes to

$Cons(c, s1, s2) \equiv c.startJobs1 \rightarrow s1$
$\star\ c.startJobs2 \rightarrow s2$.

The precondition for the `consume()` method becomes:

$Cons(c, s1, s2) \star Listseg(s1, null, 0, 10)$
$\star\, Listseg(s2, null, 0, 10)$.

Although the behaviour of the Consumer and Producer classes have not changed, the precondition for `makeActive()` in class Control does change:

$Ctrl(this, p1, p2, c1, c2) \star Prod(p1, ss1, es1, sl1, el1)$
$\star\, Prod(p2, ss2, es2, sl2, el2) \star Cons(c1, sl1, sl2)$
$\star\, Cons(c2, ss1, ss2) \star Listseg(ss1, null, 0, 10)$
$\star\, Listseg(ss2, null, 0, 10) \star Listseg(sl1, null, 0, 10)$.
$\star\, Listseg(sl2, null, 0, 10)$

The changes occur because the pointers to the job queues have been modified and the separation logic specifications have to reflect the changes. This leads to a loss of modularity.

8. Related Work

There are two main lines of research that give partial solutions for the verification of object-oriented code in the presence of aliasing: the permission-based work and the separation logic approaches.

```
public class Control {
  predicate WorkingSystem() ≡ prod1→ o1⊗ prod2→ o2
         ⊗ cons1→ o3⊗ cons2 → o4
         ⊗ share(o1) in BothInRange()
         ⊗ share(o2) in BothInRange()
         ⊗ share(o3) in ConsumeInRange(0,10)
         ⊗ share(o4) in ConsumeInRange(11,100)
      Producer prod1, prod2;
      Consumer cons1, cons2;

  public Control(Producer p1, Producer p2,
              Consumer c1, Consumer c2) {
    prod1 = p1; prod2 = p2;
    cons1 = c1; cons2 = c2; }

  public void makeActive( int i)
   share(this) in WorkingSystem() —∘
       share(this) in WorkingSystem() {
    Random generator = new Random();
    int r = generator.nextInt(4);
    if (r == 0) {prod1.produce();}
      else if (r == 1) {prod2.produce();}
        else if (r == 2) {cons1.consume();}
          else {cons2.consume();}
    if (i > 0) { makeActive(i-1);}
        }
    }
```

Figure 13. Control class

Bierhoff and Aldrich (2007) developed access permissions, an abstraction that combines typestate and object aliasing information. Developers use access permissions to express the design intent of their protocols in annotations on methods and classes. Our work is a generalization of their work, as we use object propositions to modularly check that implementations follow their design intent. The typestate (DeLine and Fähndrich 2004) formulation has certain limits of expressiveness: it is only suited to finite state abstractions. This makes it unsuitable for describing fields that contain integers and can take an infinite number of values and can satisfy various arithmetical properties. Our object propositions have the advantage that they can express predicates over an infinite domain, such as the integers.

Access permissions allow predicate changes even if objects are aliased from unknown places. States and fractions (Boyland 2003) let one capture alias types, borrowing, adoption, and focus with a single mechanism. In Boyland's work, a fractional permission means immutability (instead of sharing) in order to ensure non-interference of permissions. We use permissions to keep object propositions consistent but track, split, and join permissions in the same way as Boyland.

Boogie (Barnett et al. 2006) is a modular reusable verifier for Spec# programs. It provides design-time feedback

and generates verification conditions to be passed to an automatic theorem prover. While Boogie allows a client to depend on properties of objects that it owns, we allow a client to depend on properties of objects that it doesn't own too.

Krishnaswami and Aldrich (2005) describe System F_{own}, an extension of System F (Girard and Taylor 1989), with references and ownership for higher-order typed languages. Their type system statically enforces ownership, ensuring the encapsulation of mutable state. They do not show how the type system can be used for proving that a method satisfies its specification, while we have shown how to verify code using our methodology.

Krishnaswami et al. (2010) show how to modularly verify programs written using dynamically-generated bidirectional dependency information. They introduce a ramification operator in higher-order separation logic, that explains how local changes alter the knowledge of the rest of the heap. Their solution is application specific, as they need to find a version of the frame rule specifically for their library. Our methodology is a general one that can potentially be used for verifying any object-oriented program.

The approach developed by Jacobs and Piessens (2011) enables the verification of fine-grained concurrent modules and their clients. Their work associates a data invariant with each lock that protects shared state. This invariant is similar to the predicate invariants for share in our system, the difference being that they use it to verify concurrent programs, while we use the approach to achieve more modular reasoning and information hiding even for sequential programs that have aliasing. To verify programs, they augment them with auxiliary variables and parameters ranging over commands. We do not have to change the code in order to verify our specifications.

Nanevski et al. (2007) developed Hoare Type Theory (HTT), which is an annotated language that combines a dependently typed, higher-order language with stateful computations. While HTT offers a semantic framework for elaborating more practical external languages, our work targets Java-like languages and does not have the complexity overhead of higher-order logic.

9. Future Work

We believe our proof rules from Figure 7 are sound and we can provide an intuitive argument. In the future, we plan to formally prove soundness. A similar system of rules (Bierhoff and Aldrich 2007) has been proven to be sound. Also, we want to augment the target language with more features (for eg., inheritance) and then design formal rules to reason about verification.

References

Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, pages 364–387. Springer, 2006.

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320. ACM Press, 2007.

John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '04: European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

Dino Distefano and Matthew J. Parkinson. jstar: Towards practical verification for java. In *OOPSLA'08*. ACM, 2008.

Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI'02*. ACM, 2002.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

Y.L.J-Y Girard and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. 2001.

Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL'11*, 2011.

Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages, 2005.

Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *Types In Languages Design And Implementation*, pages 63–76, 2010.

Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. In *ESOP'07*, pages 189–204, 2007.

D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

Appendix

$$\frac{P_1; P_2 \vdash P_3}{P_1 \otimes P_2 \vdash P_3} \ (\otimes \text{L})$$

$$\frac{P_1 \vdash P_3 \qquad P_2 \vdash P_3}{P_1 \oplus P_2 \vdash P_3} \ (\oplus \text{L})$$

$$\frac{P_4 \vdash P_1 \qquad P_5 \vdash P_2}{P_4,\ P_5 \vdash P_1 \otimes P_2} \ (\otimes \text{R})$$

$$\frac{P_3 \vdash P_1;\ P_2}{P_3 \vdash P_1 \oplus P_2} \ (\oplus \text{R})$$

$$\frac{P_1(x_1) \vdash P_2 \qquad (general\ x_1)}{\exists x.P_1(x) \vdash P_2} \ (\text{PRE-}\exists)$$

$$\frac{P_1(x_1) \vdash P_2 \qquad (general\ x_1)}{\forall x.P_1(x) \vdash P_2} \ (\text{PRE-}\forall)$$

$$\frac{\exists x.P_1(x) \vdash P_2}{P_1(x_1) \vdash P_2 \quad (constructed\ x_1)} \ (\text{POST-}\exists)$$

$$\frac{\forall x.P_1(x) \vdash P_2}{P_1(x_1) \vdash P_2 \quad (general\ x_1)} \ (\text{POST-}\forall)$$

Figure 14. Rules for proving predicates

$$\frac{x_1 \leq x_2; x_2 \leq x_3; x_1 \leq x_3 \vdash P}{x_1 \leq x_2; x_2 \leq x_3 \vdash P} \ (\text{M})$$

$$\frac{a = b; c = a + d; c = b + d \vdash P}{a = b; c = a + d \vdash P} \ (\text{T})$$

Figure 15. Arithmetic rules