Object Propositions

Ligia Nistor+, Jonathan Aldrich+, Stephanie Balzer+, and Hannes Mehnert*

+School of Computer Science, Carnegie Mellon University
*IT University of Copenhagen
{lnistor,aldrich,balzers}@cs.cmu.edu, hame@itu.dk

Abstract. The presence of aliasing makes modular verification of object-oriented code difficult. If multiple clients depend on the properties of an object, one client may break a property that others depend on.
We have developed a modular verification approach based on the novel abstraction of *object propositions*, which combine 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. Our permission system allows verification using a mixture of linear and nonlinear reasoning. We thus offer an alternative to separation logic verification approaches. Object propositions can be more modular in some cases than separation logic because they can more effectively hide the exact aliasing relationships within a module. We validate the practicality of our approach by verifying an instance of the composite pattern. We implement our methodology in the intermediate verification language Boogie (of Microsoft Research), for the composite pattern example.

1 Introduction

We propose a method for modular verification of object-oriented code in the presence of aliasing, i.e., the existence of multiple references to the same object. The seminal work of Parnas [21] describes the importance of modular programming, where the information hiding criteria is used to divide the system into modules.

We introduce the notion of an *object proposition* for the modular verification of object-oriented code in the presence of aliasing. Object propositions combine abstract predicates on objects with aliasing information about the objects (represented by fractional permissions). They are associated with object references and declared by programmers as part of method pre- and post-conditions. Through the use of object propositions, we are able to hide the shared data that two objects have in common. The implementations of the two objects use fractions to describe how to access the common data, but this common data need not be exposed in their external interface. Our main contributions are:

- A verification methodology that unifies substructural logic-based reasoning with invariant-based reasoning. Linear permissions (object propositions where the fraction is equal to 1) permit reasoning similar to separation logic, while fractional permissions (object propositions where the fraction is less

than 1) introduce non-linear invariant-based reasoning. Unlike prior work [6], fractions do not indicate immutability; instead, they allow mutations that may introduce temporary inconsistency before restoring a specified invariant.
- A proof of soundness in support of the system.
- Validation of the approach by specifying and proving partial correctness of an instance of the composite pattern.
- An encoding in the intermediate verification language Boogie [2] of our methodology, for a simple example and for the composite pattern.

2 Overview

Our methodology uses abstract predicates [20] to characterize the state of an object. We embed those predicates in a logical framework, and specify sharing using *fractions* [6]. A fraction can be equal to 1 or it can be less than 1.

If in the system there is only one reference to an object, that reference has a fraction of 1 to the object, and thus full modifying control over its fields. If there are multiple references to an object, each reference has a fraction less than 1 to the object and each can modify the object as long as that modification does not break a predefined invariant (expressed as a predicate). In case that modification is not an atomic action (and instead is composed of several steps), the invariant might be broken in the course of the modification, but it must be restored at the end of the modification.

We introduce the novel *object propositions*. To express that the object q in Figure 1 has full modifying control of a queue of integers greater or equal to 0 and less than or equal to 10, we use the object proposition $q@1\ \mathit{Range}(0, 10)$. This states that there is a unique reference q pointing to a queue of integers in the range [0,10].

We want our checking approach to be modular and to verify that implementations follow their design intent. In our approach, method pre- and post-conditions are expressed using object propositions over the receiver and arguments of the method. To verify the method, the *abstract* predicate in the object proposition for the receiver object is interpreted as a *concrete* formula over the current values of the receiver object's fields. Following Fähndrich and DeLine [10], our verification system maintains a *key* for each field of the receiver object, which is used to track the current values of those fields through the method. A key $o.f \rightarrow x$ represents read/write access to field f of object o holding a value represented by the concrete value x.

As an illustrative example, we consider two linked queues q and r that share a common tail p, in Figure 1. In prior work on separation logic or dynamic frames, 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 that access the objects in the lists q and r. Using our object propositions, we have to mention only a permission $q@1\ \mathit{Range}(0, 10)$ in the specification of a method accessing q. The fact that p is shared between

the two aliases is hidden by the abstract predicate $Range(0, 10)$. In Section 4 we discuss this example in more detail.

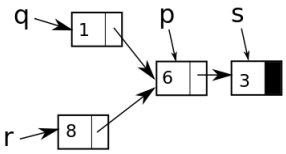

Fig. 1. Linked queues sharing the tail

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

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

  void addModulo11(int x)
   this@k Range(0, 10)  −∘  this@k Range(0, 10)
  {val = (val + x)% 11;
  if (next!=null) {next.addModulo11(x);} } }
```

Fig. 2. Link class and Range predicate

3 Current Approaches

The verification of object-oriented code can be achieved using the classical invariant-based technique [3]. When using this technique, all objects of the same class have to satisfy the same invariant. The invariant has to hold in all visible states of an object, but can be broken inside the method. Methods that can be written for each class are restricted because now each method of a particular class has to have the invariant of that class as a postcondition; the invariant of an object cannot depend on another object's state, unless additional features such as ownership [17] are added. Thus the classic technique for checking object invariants ensures that objects remain well-formed, but it does not help with reasoning about how they change over time (other than that they do not break the invariant).

Separation logic approaches [20], [9], [7], etc. bypass the limitations of invariant-based verification techniques by requiring that each method describe its footprint. Separation logic allows us to reason about how objects' state changes over time. On the downside, now the specification of a method has to reveal the structures of objects that it uses. Our methodology can be seen as an alternative to separation logic verification, that can be more modular for some examples. By encoding our verification in Boogie, we have proved that it is amenable to automation.

On the other hand, permission-based work [4], [8], [6] gives another partial solution for the verification of object-oriented code in the presence of aliasing. By using share and/or fractional permissions referring to the multiple aliases of an object, it is possible for objects of the same class to have different invariants.

Krishnaswami et al. [15] show how to modularly verify programs written using dynamically-generated bidirectional dependency information. 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.

Nanevski et al. [18] developed Hoare Type Theory (HTT), which 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.

Summers and Drossopoulou [22] introduce Considerate Reasoning, an invariant-based verification technique adopting a relaxed visible-state semantics. While their work is similar to ours in that we both allow a client to depend on properties of objects that it doesn't (exclusively) own, they differ from us because they use the classical invariant technique, with its drawbacks discussed above.

4 Example: Queues of integers

In Figure 2, we present a class that defines object propositions which are useful for reasoning about whether the implementation of a method respects its specification. Our specification logic is based on linear logic[12], a simplification of separation logic that retains the advantages of separation logic's frame rule. Object propositions are treated as resources that may not be duplicated, and which are consumed upon usage. 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).

The predicate *Range(int x, int y)* in Figure 2 ensures that all the elements in a linked queue starting from the current Link are in the range $[x, y]$. We do not need to specify $this.val$ in the definition of the predicate because $this$ is implicit for all fields of a predicate of a class. The specification of the method $addModulo11$ has as precondition $this@k\ Range(0, 10)$: the reference calling the method has to have a fraction k to the queue and it has to satisfy the $Range(0, 10)$ predicate (which is the invariant in this example). The postcondition following the $\multimap$ sign states that at the end of the method all the cells of the queue are still in the range [0,10], no matter what modifications took place inside the method. Thus if reference q of Figure 1 calls the method $addModulo11$, and after reference r calls the same method, reference r can rely on the invariant that even after q modified the queue, all the integers in the queue are still in the range [0,10].

A critical mechanism in our methodology is *packing/unpacking* [8]. When the code modifies a field, the specification has to follow suit and unpack the predicate that contains that field (unpacking a predicate gives read/write access to the fields of that predicate). At the end of a method, the fields have been modified and after checking that a predicate holds, we are allowed to pack back that predicate.

Newly created objects have a fraction of 1, and their state can be manipulated to satisfy different predicates defined in the class. At the point where the fraction to the object is first split into two fractions less than 1 (see Figure 4), the predicate currently satisfied by the object's state becomes an invariant that the object will always satisfy in future execution. Different references pointing to the

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

The specification in separation logic is more cumbersome and unable to hide shared data. To express the fact that all 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 s, such that $n_1 \leq i \leq n_2$, and the elements on the segment from s to p are in the interval $[n_1, n_2]$. If we wanted to verify the code below, we would be able to do it without revealing that queues q and r share the tail p.

```
Link s = new (Link(3, null),Range(0,10));
Link p = new (Link(6, s),Range(0,10));
Link q = new (Link(1, p),Range(0,10));
Link r = new (Link(8, p),Range(0,10));
r.addModulo11(9); q.addModulo11(7);
```

In separation logic, the natural pre- and post-conditions of the method *addModulo11* would be $Listseg(this, null, 0, 10)$. Thus, before calling `addModulo11` on r, we would 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, which causes a problem in the verification of the code above. 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 and unmodular: 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 $[0, 10]$. Abstract predicates used without fractional permissions have to reveal the exact structure of the queues. When we add the fractional permissions, we are able to hide the shared data and our work gets closer to Parkinson's concurrent abstract predicates [9] (with the added benefit of proven automation potential).

5 Grammar

The programming language that we are using is inspired by Featherweight Java [13], extended to include object propositions. We retained only Java concepts relevant to the core technical contribution of this paper, omitting features such as inheritance, casting or dynamic dispatch that are important but are handled by orthogonal techniques. We plan to focus on these features in future work.

We show the syntax of our simple class-based object-oriented language in Figure 5. In addition to the usual constructs, each class can define one or more abstract predicates Q in terms of concrete formulas R. Each method comes with pre and post-condition formulas. Formulas include object propositions P, terms,

primitive binary predicates, conjunction, disjunction, keys, and quantification. We distinguish effectful expressions from simple terms, and assume the program is in let-normal form. The pack and unpack expression forms are markers for when packing and unpacking occurs in the proof system. In the grammar, r represents a reference to an object and i represents a reference to an integer. In

$$
\begin{array}{rcl}
\mathsf{Prog} & ::= & \overline{\mathsf{ClDecl}}\ \mathsf{e} \\
\mathsf{ClDecl} & ::= & \texttt{class}\ C\ \texttt{\{}\ \overline{\mathsf{FldDecl}}\ \overline{\mathsf{PredDecl}}\ \overline{\mathsf{MthDecl}}\ \texttt{\}} \\
\mathsf{FldDecl} & ::= & \mathsf{T}\ f \\
\mathsf{PredDecl} & ::= & \texttt{predicate}\ Q\texttt{(}\overline{\mathsf{T}\ \mathsf{x}}\texttt{)} \equiv \mathsf{R} \\
\mathsf{MthDecl} & ::= & \mathsf{T}\ m\texttt{(}\overline{\mathsf{T}\ \mathsf{x}}\texttt{)}\ \mathsf{MthSpec}\ \texttt{\{}\ \overline{\mathsf{e}}\texttt{;}\ \texttt{return}\ \mathsf{e}\ \texttt{\}} \\
\mathsf{MthSpec} & ::= & \mathsf{R} \multimap \mathsf{R} \\
\mathsf{R} & ::= & \mathsf{P} \mid \mathsf{R} \otimes \mathsf{R} \mid \mathsf{R} \oplus \mathsf{R} \mid \\
 & & \exists \overline{z}.\mathsf{R} \mid \forall \overline{z}.\mathsf{R} \mid r.f \to \mathsf{x} \mid \mathsf{t}\ \mathsf{binop}\ \mathsf{t} \\
\mathsf{P} & ::= & r\texttt{@}\mathsf{k}\ Q(\overline{\mathsf{t}}) \mid \texttt{unpacked(}r\texttt{@}\mathsf{k}\ Q(\overline{\mathsf{t}})\texttt{)} \\
\mathsf{k} & ::= & \frac{n_1}{n_2}\ (\text{where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2) \\
\mathsf{e} & ::= & \mathsf{t} \mid r.f \mid r.f\ \texttt{=}\ \mathsf{t} \mid r.m(\overline{\mathsf{t}}) \mid \texttt{new}\,C(\overline{\mathsf{t}}) \mid \\
 & & \texttt{if (t) \{ e \} else \{ e \}} \mid \\
 & & \texttt{let x = e in e} \mid \\
 & & \mathsf{t}\ \mathsf{binop}\ \mathsf{t} \mid \mathsf{t}\ \texttt{\&\&}\ \mathsf{t} \mid \mathsf{t} \,\|\, \mathsf{t} \mid \texttt{!}\ \mathsf{t} \mid \\
 & & \texttt{pack}\ r\texttt{@k}\ Q(\overline{\mathsf{t}})\ \texttt{in e} \mid \\
 & & \texttt{unpack}\ r\texttt{@k}\ Q(\overline{\mathsf{t}})\ \texttt{in e} \\
\mathsf{t} & ::= & \mathsf{x} \mid n \mid \texttt{null} \mid \texttt{true} \mid \texttt{false} \\
\mathsf{x} & ::= & r \mid i \\
\mathsf{binop} & ::= & + \mid - \mid \% \mid = \mid \texttt{!=} \mid \leq \mid < \mid \geq \mid > \\
\mathsf{T} & ::= & C \mid \texttt{int} \mid \texttt{Boolean}
\end{array}
$$

Fig. 3. Language and Object Propositions Grammar

order to allow objects to be aliased, we must split a fraction of 1 into multiple fractions less than 1 [6]. When an object is created, the only reference to it has a fraction of 1. Since object propositions are considered resources, a fraction of 1 is never duplicated. We also allow the inverse of splitting permissions: joining, where we define the rules in Figure 4.

6 Proof Rules

This section describes the proof rules that can be used to verify correctness properties of code.

$$
\begin{array}{rcl}
\textit{type context}\ \Gamma & ::= & \cdot \mid \Gamma, x : T \\
\textit{linear context}\ \Pi & ::= & \bigoplus_{i=1}^{n} \Pi_i \\
\Pi_i & ::= & \cdot \mid \Pi_i \otimes \mathsf{P} \mid \Pi_i \otimes t_1\ \texttt{binop}\ t_2 \mid \\
 & & \Pi_i \otimes r.f \to x \mid \exists \overline{z}.\mathsf{P} \mid \forall \overline{z}.\mathsf{P}
\end{array}
$$

The judgment to check an expression e is of the form $\Gamma; \Pi \vdash e : \exists x.T; R$. This is read "in valid context Γ and linear context Π, an expression e executed has type T with postcondition formula R".This judgment is within a receiver class C, which is mentioned when necessary in the assumptions of the rules. By writing

$\exists x$, we bind the variable x to the result of the expression e in the postcondition. Γ gives the types of variables and references, while Π is a precondition in disjunctive normal form. The linear context Π should be just as general as R.

The static proof rules also contain the following judgments: $\Gamma \vdash r : C$, $\Gamma; \Pi \vdash R$ and $\Gamma; \Pi \vdash r.T; R$. The judgment $\Gamma \vdash r : C$ means that in valid type context Γ, the reference r has type C. The judgment $\Gamma; \Pi \vdash R$ means that from valid type context Γ and linear context Π we can deduce that object proposition R holds. The judgment $\Gamma; \Pi \vdash r.T; R$ means that from valid type context Γ and linear context Π we can deduce that reference r has type T and object proposition R is true about r.

Before presenting the detailed rules, we provide intuition for why our system is sound (the formal soundness theorem is proved in our technical report [19], Section 9.1). The soundness of the proof rules means that given a heap that satisfies the precondition formula, a program that typechecks and verifies according to our proof rules will execute, and if it terminates, will result in a heap that satisfies the postcondition formula. The first invariant enforced by our system is that there will never be two conflicting object propositions to the same object. The fraction splitting rule can give rise to only one of two situations, for a particular object: there exists a reference to the object with a fraction of 1, or all the references to this object have fractions less than 1. For the first case, sound reasoning is easy because aliasing is prohibited. The second case, concerning

$$\frac{k \in (0,1]}{r@k\ Q(\bar{t}) \vdash\ r@\tfrac{k}{2}\ Q(\bar{t}) \otimes r@\tfrac{k}{2}\ Q(\bar{t})}\ (\textsc{Split}) \qquad \left|\frac{\epsilon \in (0,1) \qquad k \in (0,1] \qquad \epsilon < k}{r@\epsilon\ Q(\overline{t_1}) \otimes r@(k-\epsilon)\ Q(\overline{t_1}) \vdash\ r@k\ Q(\overline{t_1})}\ (\textsc{Add})\right.$$

Fig. 4. Rules for adding/splitting fractions

fractional permissions less than 1, follows an inductive argument in nature. The argument is based on the property that the invariant of a shared object (one can think of an object with a fraction less than 1 as being shared) is assumed to hold whenever that object is packed.

The reader must pay attention here: we *assume* that the invariant holds, we do not state that the invariant is true in the Boolean sense. This is because another reference might be in the process of modifying the same object. Even so, that reference will restore the invariant when it is done modifying the object and it will pack back the invariant. That is why we can *assume* that the invariant holds. In this way, a predicate is *true* in the Boolean sense when its definition is true and all predicates of other objects that it transitively depends on are packed. The base case in the induction occurs when an object with a fraction of 1, whose invariant holds, first becomes shared. In order to access the fields of an object, we must first unpack the corresponding predicate; by induction, we can assume its invariant holds as long as the object is packed. We know the object is packed immediately before the unpack operation, because the rules of our system ensure that a given predicate over a particular object can only be unpacked once; therefore, we know the object's invariant holds. Assignments to the object's fields may later violate the invariant, but in order to pack the object

back up we must restore its invariant. For a shared object, packing must restore the same predicate the object had when it was unpacked; thus the invariant of an object never changes once that object is shared, avoiding inconsistencies between aliases to the object. (Note that if at a later time we add the fractions corresponding to that object and get a fraction of 1, we will be able to change the predicates that hold of that object. But as long as the object is shared, the invariant of that object must hold.)

This completes the inductive case for soundness of shared objects. The induction is done on the steps when a predicate is packed or unpacked. All of the predicates we might infer will thus be sound because we will never assume anything more about that object than the predicate invariant, which should hold according to the above argument.

In the following paragraphs, we describe the most interesting proof rules while inlining the rules in the text. The rest of the rules are described in the technical report [19] in Section 6. In the rules below we assume that there is a class C that is the same for all the rules.

NEW checks object construction. We get a key for each field and the remaining linear context Π_1. The context Π_1 contains the object propositions of Π from which we extracted the object propositions of the form $z.f \to t$ containing the fields of the newly created object.

$$\frac{fields(C) = \overline{T\ f} \qquad \Gamma \vdash \overline{t : T}}{\Gamma; \Pi \vdash \texttt{new}\ C(\overline{t}) : \exists z.C; z.\overline{f} \to \overline{t} \otimes \Pi_1}\ \textsc{New}$$

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 judgment MTYPE) and then goes on to state the object proposition holding for the result. The $\vdash$ notation in the fourth premise of the CALL rule represents entailment in linear logic.

The reader might see that there are some concerns about the modularity of the CALL rule: Π_1 shouldn't contain unpacked predicates. Indeed, it is important that the CALL rule tracks all shared predicates that are unpacked. It does not track predicates that are packed, nor unpacked predicates that have a fractional permission of 1. Our verification methodology works best when the predicates of shared objects being passed to methods are all packed. The normal situation is indeed that all shared predicates are packed, and any method can be called in this situation. We only make calls with a shared unpacked predicate when traversing a data structure hand-over-hand as in the Composite pattern in Section 7. The fact that we need to track unpacked shared predicates does represent a limitation in our system, however, it is one that goes hand in hand with the advantage of supporting shared predicates. The implementation in Boogie [2] that we describe in Section 8 has offered us insight in how to deal with this situation in a practical way.

$$\frac{\begin{array}{c}\Gamma \vdash r_0 : C_0 \quad \Gamma \vdash \overline{t_1 : T} \\ \Gamma; \Pi \vdash [r_0/this][\overline{t_1}/\overline{x}]R_1 \otimes \Pi_1 \\ mtype(m, C_0) = \forall \overline{x : T}.\exists result.T_r; R_1' \multimap R \\ R_1 \vdash R_1' \\ \Pi_1 \textit{ cannot contain unpacked predicates}\end{array}}{\Gamma; \Pi \vdash r_0.m(\overline{t_1}) : \exists\ result.T_r; [r_0/this][\overline{t_1}/\overline{x}]R \otimes \Pi_1}\ \textsc{Call}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash t_1 : T_i; t_1@k_0\ Q_0(\overline{t_0}) \otimes \Pi_1 \\ \Gamma; \Pi_1 \vdash r_1.f_i : T_i; r_i'@k'\ Q'(\overline{t'}) \otimes \Pi_2 \\ \Pi_2 \vdash r_1.f_i \to r_i' \otimes \Pi_3\end{array}}{\begin{array}{c}\Gamma; \Pi \vdash r_1.f_i = t_1\ : \exists x.T_i; x@k'\ Q'(\overline{t'}) \otimes t_1@k_0\ Q_0(\overline{t_0}) \\ \otimes\ \ r_1.f_i \to t_1 \otimes \Pi_3\end{array}}\ \textsc{Assign}$$

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, thus giving us permission to modify the fields. The rules for packing and unpacking are Pack1, Pack2, Unpack1 and Unpack2. As mentioned before, when we pack an object to a predicate with a fraction less than 1, we have to pack it to the same predicate that was true before the object was unpacked. The restriction is not necessary for a predicate with a fraction of 1: objects that are packed to this kind of predicate can be packed to a different predicate than the one that was true for them before unpacking.

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; [\overline{t_2}/\overline{x}]R_2 \otimes \Pi_1 \\ \texttt{predicate}\ Q_2(\overline{Tx}) \equiv R_2 \in C \\ \Gamma; (\Pi_1 \otimes r@1\ Q_2(\overline{t_2})) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \vdash \texttt{pack}\ r@1\ Q_2(\overline{t_2})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack1}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; [\overline{t_1}/\overline{x}]R_1 \otimes \texttt{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \quad 0 < k < 1 \\ \Gamma; (\Pi_1 \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \vdash \texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack2}$$

As mentioned earlier, we allow unpacking of multiple predicates, as long as the objects don't alias. We also allow unpacking of multiple predicates of the same object, because we have a single linear write permission to each field. There can't be any two packed predicates containing write permissions to the same field.

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; r@1\ Q(\overline{t_1}) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \\ \Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \vdash \texttt{unpack}\ r@1\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Unpack1}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; r@k\ Q(\overline{t_1}) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \quad 0 < k < 1 \\ \Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes \texttt{unpacked}(r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R \\ \forall r', \overline{t} : (\ \texttt{unpacked}(r'@k'\ Q(\overline{t})) \in \Pi \Rightarrow \Pi \vdash r \neq r')\end{array}}{\Gamma; \Pi \vdash \texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Unpack2}$$

We have also developed rules for the dynamic semantics, that are used in proving the soundness of our system, together with the standard rules of linear logic and integer arithmetic. The reader can refer to the additional technical report [19], Section 9, for the dynamic semantics rules and proof of soundness.

7 Composite

The Composite design pattern [11] expresses the fact that clients treat individual objects and compositions of objects uniformly. Verifying implementations of the Composite pattern is challenging, especially when the invariants of objects in the tree depend on each other [16], and when interior nodes of the tree can be modified by external clients, without going through the root. As a result, verifying the Composite pattern is a well-known challenge problem, with some attempted solutions presented at SAVCBS 2008 (e.g. [5, 14]). We describe a new formalization and proof of the Composite pattern using object propositions that provides more local reasoning than prior solutions. For example, in Jacobs et al. [14] a global description of the precise shape of the entire Composite tree must be explicitly manipulated by clients; in our solution a client simply has a fraction to the node in the tree it is dealing with.

We implement a popular version of the Composite design pattern, as an acyclic binary tree, where each Composite has a reference to its left and right children and to its parent. The code is given in Figure 5.

Each Composite caches the size of its subtrees in a count field, so that a parent's count depends on its children's count. Clients can set a new left child or right child at any time, to any node. This operation changes the count of all ancestors, which is done through a recursive call of the method $updateCountRec()$ that starts a notification protocol from the current node and up the tree to the root. The pattern of circular dependencies and the notification mechanism are hard to capture with verification approaches based on ownership or uniqueness. We assume that the notification terminates (that the tree has no cycles) and we verify that the Composite tree is well-formed: parent and child pointers line up and counts are consistent.

Previously the Composite pattern has been verified with a related approach based on access permissions and typestate [5]. That verification abstracted counts to an even/odd typestate and relied on non-formalized extensions of a formal system.

7.1 Specification

A Composite tree is well-formed if the field count of each node n contains the number of nodes of the tree rooted in n. A node of the Composite tree is a leaf when the left and right fields are null.

The goal of the specification is to verify that after we change the left child (or right) of a node by calling the method $setLeft()$ (or $setRight()$), the tree is still in a consistent state. Since the count field of a node depends on the count fields

```
public Composite()
{this.count = 1;
this.left = null;
this.right = null;
this.parent = null; }

private void updateCountRec(){
if (this.parent != null)
 {this.updateCount();
  this.parent.updateCountRec();}
else this.updateCount(); }

private void updateCount(){
int newc = 1;
if (this.left != null)
  newc = newc + left.count;
if (this.right != null)
  newc = newc + right.count;
this.count = newc; }

public void setLeft(Composite l)
{if (l.parent==null){
l.parent= this;
this.left = l;
this.updateCountRec(); }}

public void setRight(Composite r)
{if (r.parent==null){
r.parent = this;
this.right = r;
this.updateCountRec(); }}
```

Fig. 5. Composite class

$$\begin{array}{l}
\mathsf{predicate}\ \mathit{count}\ (\mathsf{int\ c}) \equiv \\
\exists\ \mathsf{ol,\ or,\ lc,\ rc.\ this.count} \rightarrow \mathsf{c} \otimes \\
\mathsf{c} = \mathsf{lc} + \mathsf{rc} + 1\ \otimes \mathsf{this}@\frac{1}{2}\ \mathit{left}(\mathsf{ol,\ lc}) \\
\otimes\ \mathsf{this}@\frac{1}{2}\ \mathit{right}(\mathsf{or,\ rc}) \\
\\
\mathsf{predicate}\ \mathit{left}\ (\mathsf{Composite\ ol, int\ lc}) \equiv \\
\mathsf{this.left} \rightarrow \mathsf{ol} \otimes \big((\mathsf{ol} = \mathsf{null}\ \multimap\ \mathsf{lc} = 0) \\
\oplus\ (\mathsf{ol} \neq \mathsf{null}\ \multimap\ \mathsf{ol}@\frac{1}{2}\ \mathit{count}(\mathsf{lc}))\big) \\
\\
\mathsf{predicate}\ \mathit{right}\ (\mathsf{Composite\ or, int\ rc}) \equiv \\
\mathsf{this.right} \rightarrow \mathsf{or} \otimes \big((\mathsf{or} = \mathsf{null}\ \multimap\ \mathsf{rc} = 0) \\
\oplus\ (\mathsf{or} \neq \mathsf{null}\ \multimap\ \mathsf{or}@\frac{1}{2}\ \mathit{count}(\mathsf{rc}))\big) \\
\\
\mathsf{predicate}\ \mathit{parent}\ () \equiv \\
\exists \mathsf{op,\ c,\ k.\ this.parent} \rightarrow \mathsf{op} \otimes \\
\mathsf{this}@\frac{1}{2}\ \mathit{count}(\mathsf{c}) \otimes \\
\Big((\mathsf{op} \neq \mathsf{null}\ \multimap\ \mathsf{op}@\mathsf{k}\ \mathit{parent}() \otimes \\
(\mathsf{op}@\frac{1}{2}\ \mathit{left}(\mathsf{this,\ c}) \oplus \mathsf{op}@\frac{1}{2}\ \mathit{right}(\mathsf{this,\ c}))\big) \\
\oplus\ (\mathsf{op} = \mathsf{null}\ \multimap\ \mathsf{this}@\frac{1}{2}\ \mathit{count}(\mathsf{c}))\Big)
\end{array}$$

Fig. 6. Predicates for Composite

of its children nodes, we must ensure that after modifying a child the invariants of the transitive parents are restored.

We use the following methodology for verification: each node has a fractional permission to its children, and each child has a fractional permission to its parent. We allow unpacking of multiple object propositions as long as they satisfy the heap invariant: if two object propositions are unpacked and they refer to the same object then we require that they do not have fields in common. For more information about the heap invariants, see our technical report [19] Section 9.

As a downside, the specification of the composite is verbose: we have four predicates that are recursive and depend on each other. The source of this verbosity comes from the the fact that the composite example itself is complicated and thus necessitates a complicated specification and verification. We allow clients to directly mutate any place in the tree, using predicates that reason about one object in the composite at a time and treat other objects in the

composite abstractly. Note that a simpler specification is possible in our system but would limit mutation to the root of the tree.

The predicates of the Composite class are presented in Figure 6. The definition of each predicate mentions the field with the same name and how that field interacts with the other predicates. Thus, the predicate *count* has a parameter c, which is an integer representing the value at the count field. There are two existentially quantified variables lc and rc, for the *count* fields of the left child lc and the right child rc. By $c = lc + rc + 1$ we make sure that the count of *this* is equal to the sum of the counts for the children plus 1. By $\mathsf{this}@\frac{1}{2}\ left(ol, lc) \otimes \mathsf{this}@\frac{1}{2}\ right(or, rc)$ we connect lc to the left child (through the $left$ predicate) and rc to the right child (through the *right* predicate). The *count* predicate ensures that the tree starting at the current node has the *count* fields of all nodes in a consistent state.

The predicate $left$ states that the predicate $count(lc)$ holds for $this.left$, the left child of *this*. The predicate *right* states that the predicate $count(rc)$ holds for $this.right$, the right child of *this*. The permission for the $left$ ($right$) predicate is split in equal fractions between the *count* predicate and the left (right) child's *parent* predicate.

Inside the *parent* predicate of *this*, there is a $\frac{1}{2}$ permission to the *count* predicate (and implicitly to its *count* field) of *this*. When a method needs to modify the *count* field of an object, it will need a fraction of 1 to the *count* predicate, since this predicate has parameters in its declaration and the changes of these parameters are visible to other references. The other $\frac{1}{2}$ permission is taken from unpacking the $left$ predicate (or *right*, depending if *this* is the left or right child of its parent) of *op*. This is the reason why there is only a half permission to the *count* predicate in the $left$ predicate, because the other half is in the *parent* predicate. The *parent* predicate contains only a fraction of k to the parent of *this* so that any client can use the remaining fraction to reference the node and add children to the parent. Note that a client cannot use a fraction of 1 to the *parent* predicate of *this* because after the Composite tree is created and all the predicates established, the k fraction to the *parent* predicate of *this* has been used; the verification system keeps track of the fractional permissions and the clients can use that information. A client can actually use this to update the parent field, but in order to pack the *parent* predicate the client has to ensure that the field count of each node n contains the number of nodes of the tree rooted in n (the well-formedness condition of the Composite example). If this condition is not met, the client will not be able to pack the *parent* predicate; the Boogie implementation will not allow the *parent* predicate to be packed because its definition is not satisfied.

The *parent* predicate is the invariant in the Composite example and ensures that *all* the nodes in the tree, both below and above the current node, are in a consistent state. If the left child of *this* is replaced with a new node (by calling the method $setLeft$), we need to change the *count* field of *this*. Because the *count* predicate has parameters that might change when the left child of *this* is modified, we need a fraction of 1 (full permission) in order to change it. The

only invariant in the Composite example is the predicate *parent* which has no parameters; this absence of parameters makes it possible to not reveal to outside clients the changes in the *count* fields inside the tree. Other clients that depend on the *parent* invariant of any node in the tree will be able to still rely on that invariant at the end of calling the public method $setLeft$. Note that the implementation of $setLeft(l)$ does nothing in the case that parameter l already has a parent. Only the methods $setLeft$, *setRight* and the constructor in Figure 5 are public and these are the only methods that can be called by external clients; all other methods are private, as they are helper methods that help to restore the consistency in the tree and they can only be called by references internal to the tree. Thus when we obtain a full permission to the *count* field of *this* we are sure that no other reference exists to this field (internal or external).

A permission of $\frac{1}{2}$ to the *count* field of *this* is acquired by unpacking the *count* predicate of *this*. Getting the other half requires us to unpack the *parent* predicate of *this*, which gives us access to the *count* predicate of the parent *op* of *this*. Now we can unpack the *count* predicate of the parent *op* and we get access to the $left$ and *right* predicates of the parent *op*. We assume that *this* is the right child of its parent (the other way is analogous). Inside the *right* predicate of the parent, there is the other half of the permission to the *count* predicate of *this* (and implicitly to the *count* field of *this*). By adding the two halfs we have a permission of 1 to the *count* field of *this* and we can modify it by calling the method *updateCount*. We recursively unpack the *count* predicates of the ancestors of *this* all the way to the root node.

Note that after calling the method *updateCount*, the *count* predicate of *this* can be packed because the tree that has *this* as the root is consistent now. The *parent* predicate of *this* cannot be packed however because the *parent* predicate of *this* is now inconsistent. The *parent* predicates will be recursively unpacked before calling the method *updateCountRec* and they will be packed back only when the recursion finishes. Thus, the *parent* predicate of *this* will be packed only *after* the call *this.updateCountRec*() returns. Since all *parent* predicates will be packed, this signals that the tree is in a consistent state. The complete specification for each method is given in Figure 7. The method $setLeft$ (or *setRight*) is the one being called by clients when they want to modify the Composite tree and this method has to preserve the invariant *parent* in its specification. When the programmer writes the specifications of the methods *updateCount* and *updateCountRec*, he/she should be guided by what object propositions hold before the calls to these functions and what object propositions should hold afterwards, in order for the invariant *parent* to hold at the end of the method $setLeft$. The constructor of the class Composite returns half of the permission for the $left$ and *right* predicates, and half of a permission to the *parent* predicate. Note that it could return half of a permission to its *count* predicate, depending if the programmer needs that predicate to prove a property right after a new Composite object is created.

The method *updateCountRec*() takes in a fraction of $k1$ to the unpacked *parent* predicate and a half fraction to the unpacked *count* predicate of *this*,

```
public Composite()
⊸ this@½ parent() ⊗ this@½ left(null, 0) ⊗
this@½ right(null, 0)
{...}

private void updateCount()
∃ c, c1, c2, ol, or.
unpacked(this@1 count(c)) ⊗
this@½ left(ol, c1) ⊗ this@½ right(or, c2)
⊸ ∃ c. this@1 count(c)
{...}

public void setLeft(Composite l)
this ≠ l ⊗ this@½ left(null, 0) ⊗ ∃ k1, k2.
(this@k1 parent() ⊗ l@k2 parent()) ⊸
∃ k.this@k parent()
{...}
```

```
private void updateCountRec()
∃ k1, opp, lcc, k, ol, lc, or, rc.
(unpacked(this@ k1 parent()) ⊗
this.parent → opp ⊗
opp ≠ this ⊗
( ( ( ( opp ≠ null ⊸ opp@k parent() ⊗
   ( opp@½ left(this, lcc) ⊕
   opp@½ right(this, lcc) )
) ) ⊕
(opp = null ⊸ this@½ count(lcc) )
) ) ⊗
unpacked(this@½ count(lcc)) ⊗
this@½ left(ol, lc) ⊗this@½ right(or, rc)
⊸ ∃ k1.this@k1 parent())
{...}
```

Fig. 7. Specifications for Composite methods

and it returns the $k1$ fraction to the packed *parent* predicate. This means that after calling this method, the *parent* predicate holds for *this*.

In the same way, the method *updateCount* takes in the unpacked predicate *count* for *this* object and it returns the *count* predicate packed for *this*. The object propositions this@½ left(ol, c1) ⊗this@½ right(or, c2) come from the definition of the unpacked predicate count(c), they are not different ones. The only part of the predicate $count(c)$ that is not in the precondition of the method *updateCount* is $c = lc + rc + 1$; this is because when entering the method *updateCount*, the *count* field of *this* might not be in a consistent state, considering that the left (or right) child of *this* has been replaced in *setLeft* (or *setRight*).

Thus, after calling *updateCount*(), the object *this* satisfies its *count* predicate. We need a fraction of 1 to the *count* predicate both in the precondition and the postcondition because the method *updateCount* modifies the field *count* of *this* and because the parameter of the predicate *count* is the actual value of the field *count*. If this value is modified and revealed to other references, the method modifying it should have a permission of 1 (full) to the field *count*.

The method $setLeft(Composite\ l)$ takes in a fraction to the *parent* predicate of *this* and a fraction to the *parent* predicate of l . The postcondition shows that after calling $setLeft$, the *parent* predicate holds.

8 Implementation of Composite using Boogie

We manually verified the Composite example (see Section 11.3 of our technical report [19]) and we implemented our verification in the intermediate verification language Boogie (see the code in Section 11.2 of our technical report). All three methods and the constructor of the Composite class from Figure 5 were formally verified by the Boogie tool [1]. In our Boogie encoding, we created a type *type Ref* to represent references of type Composite. We represented the heap by creating

maps from objects to their fields: for example we represented the field $left$ by *var left: [Ref]Ref;* which maps an object of type Composite to its left child of type Composite. We created a new map type to keep count of fractions *type FractionType = [Ref, PredicateTypes] int;*. Given a reference of type Composite and the name of a predicate, a map of type *FractionType* returns the fraction associated with that reference and that predicate. In our Boogie encoding, a fraction of 1 is represented by 100, while a fraction of $\frac{1}{2}$ by 50. We used *assume* statements in Boogie to assume facts that we knew were true according to our methodology. We used *assert* statements in Boogie whenever we needed to prove something(e.g. before packing a predicate).

For each predicate we wrote a function and several axioms related to that function. These axioms were of two types: related to the packing of that predicate - stating what are the properties necessary for packing that predicate and for it being true; and related to the unpacking of that predicate - given that the predicate is true, we stated the properties that are true according to the definition of the predicate.

Since Boogie creates verification conditions that it sends to the $Z3$ theorem prover, we had to pay special attention to existential and universal quantification. We wrote three axioms that helped our proof with the instantiation of variables. For example, the parameter c of the *count* predicate represents the value of the *count* field of *this*, but in the *parent* predicate it is existentially quantified. We wrote an axiom that indicates to Boogie that the existentially quantified value c is actually $count[this]$, *i.e.*, the value of the *count* field of *this*. We also had to write two frame axioms that informed Boogie that even if a global map was modified, that did not impact the part of the global map that was used in certain predicates and thus the predicates were not modified.

The most interesting insight that we got from using Boogie for the verification of the Composite pattern was that when we enter a method with some predicates unpacked (in the precondition), as in the case of the method *updateCountRec*, we cannot assume that the invariants that are packed are true in the Boolean sense. This is related to the discussion of the CALL rule from Section 6. We can however assume that they *hold*, which means that they will be true at the end of the method that is accessing them. If a predicate is true or not in the Boolean sense does not modify the fractions to other objects that it holds inside. We can use this information about fractions to obtain full permission to the predicates that we want to modify (such as the predicate *count*, as described in the previous section).

Our final goal is to create a tool that automatically translates our Java-like code and specifications into Boogie. We believe that most of the Boogie encoding that we have manually translated can be automatically translated into Boogie, apart from the axioms about the instantiation of existential variables and the frame axioms. Without these axioms, Boogie will report that some assertions might not hold. In that case, the developers could simply assume those statements instead of trying to prove them using *assert*, or they could improve the translation by writing the axioms themselves.

References

1. http://rise4fun.com/Boogie/.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387. Springer, 2005.
3. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology Special Issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, 3(6):27–56, June 2004.
4. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007.
5. Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the composite design pattern. In *Proc of SAVCBS 2008*, 2008.
6. John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003.
7. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.
8. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
9. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
10. Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
12. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.
13. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. pages 132–146, 2001.
14. Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Proc of SAVCBS 2008*, 2008.
15. Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI '10*, pages 63–76, 2010.
16. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19:159–189, June 2007.
17. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *In ECOOP*, 2004.
18. Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *ESOP, volume 4421 of LNCS*, pages 189–204, 2007.
19. Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. Object propositions. Technical Report CMU-CS-13-132, Carnegie Mellon University, 2013. http://www.cs.cmu.edu/~lnistor/techReportCMU-CS-13-132.pdf.
20. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
21. D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.
22. Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design patterns. In *VMCAI*, volume 5944 of Lecture Notes in Computer Science, pages 328–344, 2010.