The Implementation of Object Propositions: the Oprop Verification Tool

Ligia Nistor (0000-0002-4714-5034) and Jonathan Aldrich (0000-0003-0631-5591)

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

Abstract. In this paper we present Oprop, a tool that implements the theory of object propositions. We have recently introduced object propositions as a modular verfication technique that combines abstract predicates and fractional permissions. The Oprop tool, found as a web application at *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com*, verifies programs written in a simplified version of Java augmented with the object propositions specifications. Our tool parses the input files and automatically translates them into the intermediate verification language Boogie, which is verified by the Boogie verifier that we use as a back end. We present the details of our implementation, the rules of our translation and how they are applied on an example. We describe an instance of the challenging Composite design pattern, that we have automatically verified using the Oprop tool, and prove the equivalence between formulas in Oprop and their translation into Boogie.

Keywords: Formal · Verification · Oprop · Object · Propositions · Linear · Logic · Java · Boogie · Translation · Equivalence · Proof · Tool · Web · Service

1 Motivation

In this paper we present a practical verification tool, Oprop, for object-oriented programs in single-threaded settings. In 2014 we published a method for modular verification [23] of object-oriented code in the presence of aliasing. Our approach, *object propositions*, builds on separation logic and inherits its modularity advantages, but provides additional modularity by allowing developers to hide shared mutable data that two objects have in common. The implementations of the two objects have a shared fractional permission [5] to access the common data (for example each of the two objects has a field pointing to the same object and thus each object has a half fraction to that object), but this need not be exposed in their external interface.

Our work is modular to a class: in each class we define predicates that objects of other classes can rely on. We get the modularity advantages while also supporting a high degree of expressiveness by allowing the modification of multiple objects. Like separation logic and permissions, but unlike conventional object invariant and ownership-based work (including [20] and [21]), our system allows "ownership transfer" by passing unique permissions around (permissions with a fraction of 1). Unlike separation logic and permission systems, but like object invariant work and its extensions (for example, the work of Summers and Drossopoulou [27]), we can modify objects without owning them. More broadly, unlike either ownership or separation logic systems, in our system object A can depend on an invariant property of object B even when B is not owned by A, and when A is not "visible" from B. This has information-hiding and system-structuring benefits. Part of the innovation is combining the two mechanisms above so that we can choose between one or the other for each object, and even switch between them for a given object. By being able to formally verify different classes and methods of those classes independently of each other and automatically using our Oprop tool, our work can be used to formally verify component-based software.

The contributions of this paper are a description of how we implemented our methodology in the Oprop tool, a proof of soundness of our core translation technique, and experience verifying the Composite pattern and a few smaller examples with the tool. In Section 2 we give a background presentation of object propositions, together with an example class written in the Oprop language. Section 3 presents the tool and how it can be accessed as a web application. Section 4 and 5 present an intuitive description of the formal translation rules from Oprop into Boogie and the formal proof of equivalence between formulas written in Oprop and their Boogie translation. We conclude by showing the examples that we automatically verified using Oprop and compare our work to existing approaches, showing how our work contributes to the state of the art in the area of tool based approaches that facilitate the application of formal methods for component based software.

2 The Theory of Object Propositions

The object proposition methodology [23] uses abstract predicates [24] to characterise the state of an object, embeds those predicates in a logical framework and specifies sharing using fractional permissions [5]. When an object a has a fractional permission to object b it means that one of the fields of a is a reference to b. Object propositions are associated with object references in the code. Programmers can use them in writing method pre- and post-conditions and in the packing/unpacking annotations that they can insert in the code as part of verification.

To verify a 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 (including for fields of primitive type *int*). Following Fähndrich and DeLine [9], 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.

To gain read or write access to the fields of an object, we have to *unpack* it [8]. After a method finishes working with the fields of a shared object (an object for which we have a fractional permission, with a fraction less than 1) our proof rules ensure that the same predicate as before the unpacking holds of that shared object. If the same predicate holds, we are allowed to pack back the shared object to that predicate. Since for an object with a fractional permission of 1 there is no risk of interferences, we don't require packing to the same predicate for this kind of objects.

Object propositions are unique in providing a separation logic with fractions, in which developers can unpack an object that is shared with a fractional permission, modify its fields, and pack it again as long as the new field values validate the original abstract predicate. 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. Since they contain fractional permissions which represent resources that have to be consumed upon usage, object propositions are consumed upon usage and their duplication is forbidden. Therefore, we use linear logic [11] to write the 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).

$$
\begin{array}{rcl}
\mathsf{Prog} & ::= & \overline{\mathsf{ClDecl}}\ \mathsf{e} \\
\mathsf{ClDecl} & ::= & \texttt{class}\ C\ \{\ \overline{\mathsf{FldDecl}}\ \overline{\mathsf{PredDecl}}\ \overline{\mathsf{MthDecl}}\ \} \\
\mathsf{FldDecl} & ::= & \mathsf{T}\ f \\
\mathsf{PredDecl} & ::= & \texttt{predicate}\ Q(\overline{\mathsf{T\ x}}) \equiv \mathsf{R} \\
\mathsf{MthDecl} & ::= & \mathsf{T}\ m(\overline{\mathsf{T\ x}})\ \mathsf{MthSpec}\ \{\ \bar{\mathsf{e}};\ \texttt{return}\ \mathsf{e}\ \} \\
\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 \mathsf{x{:}T.R} \mid \exists \mathsf{z{:}double.R} \mid \exists \mathsf{z{:}double.z\ binop\ t} \Rightarrow \mathsf{R} \mid \\
 & & \forall \mathsf{x{:}T.R} \mid \forall \mathsf{z{:}double.R} \mid \forall \mathsf{z{:}double.z\ binop\ t} \Rightarrow \mathsf{R} \mid \\
 & & \mathsf{t\ binop\ t} \Rightarrow \mathsf{R} \\
\mathsf{P} & ::= & r\#\mathsf{k}\ Q(\bar{\mathsf{t}}) \mid \texttt{unpacked}(r\#\mathsf{k}\ Q(\bar{\mathsf{t}})) \mid \\
 & & r.f \rightarrow \mathsf{x} \mid \mathsf{t\ binop\ t} \\
\mathsf{k} & ::= & \frac{n_1}{n_2}\ (\text{where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2) \mid \mathsf{z} \\
\mathsf{e} & ::= & \mathsf{t} \mid r.f \mid r.f\ \texttt{=}\ \mathsf{t} \mid r.m(\bar{\mathsf{t}}) \mid \\
 & & \texttt{new}\ C(Q(\bar{\mathsf{t}})[\bar{\mathsf{t}}])(\bar{\mathsf{t}}) \mid \\
 & & \texttt{if (t) \{ e \} else \{ e \}} \mid \texttt{let x = e in e} \mid \\
 & & \mathsf{t\ binop\ t} \mid \mathsf{t}\ \texttt{\&\&}\ \mathsf{t} \mid \mathsf{t} \parallel \mathsf{t} \mid \texttt{!}\ \mathsf{t} \mid \\
 & & \texttt{pack}\ r\#\mathsf{k}\ Q(\bar{\mathsf{t}})[\bar{\mathsf{t}}]\ \texttt{in}\ \mathsf{e} \mid \texttt{unpack}\ r\#\mathsf{k}\ Q(\bar{\mathsf{t}})[\bar{\mathsf{t}}]\ \texttt{in}\ \mathsf{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 != \mid \leq \mid < \mid \geq \mid > \\
\mathsf{T} & ::= & C \mid \texttt{int} \mid \texttt{boolean} \mid \texttt{double}
\end{array}
$$

Fig. 1. Grammar of object propositions

We show the syntax of our simple class-based object-oriented language, that we call the Oprop language, in Figure 1. In rule ClDecl each class can define one or more abstract predicates Q in terms of concrete formulas R. Each method in the rule $\mathsf{MthDecl}$ comes with pre- and post-condition formulas. Formulas R include object propositions P , terms $\mathtt{t}$, primitive binary predicates, conjunction, disjunction, keys $r.f \to \mathsf{x}$, 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 an integer. The variable z represents a metavariable for fractions and it has type *double*. In a program, a fraction can be a constant of type double or it can be represented by a metavariable. The horizontal bar above a symbol means that there could be one or more occurrences of that symbol.

The example `SimpleCell.java` in Figure 2 shows an Oprop class. We created the SimpleCell example to illustrate how we can modify an object even if we do not have a full permission to it. Also, we can rely only on the specifications of methods in order to reason about them, which strengthens our modularity claim.

Note that even though the class is written in the Oprop language, the extension of the file remains *.java*. This example differs from our grammar in a couple of ways: every Oprop input file has to have the declaration of the enclosing package as first statement, and the linear conjunction and disjunction that we use in our formal grammar in Figure 2 are replaced by `&&` and `||` in the Oprop code.

Figure 2 shows the declaration of the `SimpleCell` class, the declaration of the predicates, the `changeVal` method and its specification, and the `main()` method. When the object `c` is created in the `main()` method, line 17, we have to specify the predicate that holds for it in case the object becomes shared in the future. Since the predicate `PredVal` defined on line 5 has one existentially quantified variable and the Boogie tool cannot successfully instantiate existential variables, we give the witness 2 for the variable `int v` existentially quantified in the body of the predicate `PredVal`. In general, whenever there is an existential Oprop statement in the code, we pass the witnesses for that statement explicitly. The tilde sign in the specification of the `changeVal` method on line 9 is there to differentiate between variables `k` used for fractions and other variables that are used as parameters to predicates.

When we unpack a predicate, as we do on line 12, we check that the provided witness is the right one; we do not assume that the programmer provided the right witness. We implemented the translation strategy in this way because the programmer might make a mistake and provide the wrong witness.

3 The Oprop Tool

The Oprop tool, found at *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com* as a web application, takes as input any number of files written in Java and annotated with object propositions specifications. The tool produces for each input file the corresponding Boogie translation file. We have written in Java the translation rules from Oprop into the Boogie language and we are deploying them in the form of a jar file on the Oprop web application. If the user has

```
package x;
class SimpleCell {
 int val;
 SimpleCell next;
 predicate PredVal() = exists int v : this.val -> v && v<15
 predicate PredNext() = exists SimpleCell obj :
  this.next -> obj && (obj#0.34 PredVal())
 void changeVal(int r)
 ~double k : requires (this#k PredVal()) && (r<15)
 ensures this#k PredVal()
 {
  unpack(this#k PredVal())[this.val];
  this.val = r;
  pack(this#k PredVal())[r];
 }
 void main() {
  SimpleCell c = new SimpleCell(PredVal()[2])(2, null);
  SimpleCell a = new SimpleCell(PredNext()[c])(2, c);
  SimpleCell b = new SimpleCell(PredNext()[c])(3, c);
  unpack(a#1 PredNext())[c];
  unpack(b#1 PredNext())[c];
  c.changeVal(4);
 }
}
```

Fig. 2. SimpleCell.java

provided multiple files as input, there will be multiple files produced as output. In the background, the tool concatenates the multiple files into a single one. The concatenation of the translated input files will be accessible to the user on the last page of the web application, under the link *inputBoogie.bpl*. The final result of the verification - the Boogie tool run on the*inputBoogie.bpl* file - is accessible by clicking on the link *result.txt*. The user will be able to see if the original Java file augmented with the object propositions annotations was verified or not. If an error message is displayed, the user has the option of going back to the original Java file and adding more annotations that might help the formal verification and then uploading the new file to restart the process.

The translation part of the Oprop tool is composed of two parts: JExpr [2] and the Boogie translation. JExpr is a parser for a very small subset of Java. We took this off-the-shelf parser and added support for the parsing of object propositions annotations. The JExpr system consists of the following components: a JavaCC parser, a set of Abstract Syntax Tree classes, a Contextual Semantic Analysis visitor and a type resolution class used by the Contextual Visitor. We implemented the Boogie translation in a file called *BoogieVisitor.java*. In this file we implement the translation rules presented in Section 4. By implementing the visitor design pattern, we visit all the nodes of the Abstract Syntax Tree and perform the translation for each one. We have nodes such as *FieldDeclaration*, *PredicateDeclaration*, *MethodDeclaration*, *ObjectProposition*, *FieldSelection*, *MethodSelection*, *BinaryExpression*, *FormalParameter*, *AllocationExpression*, etc.

The resulting Boogie translation is fed into the Boogie verifier, in the background of our web application. The Boogie verifier uses the Z3 high-performance theorem prover [7] to answer whether the program was verified successfully or not. The Z3 theorem prover works very well for our methodology, since our abstract predicates use integers to express the properties of fields. Reasoning about integers in an automated way can be difficult, but Z3 is one of the most prominent satisfiability modulo theories (SMT) solvers and it is used in related tools such as Boogie [3], Dafny [17], Chalice [16] and VCC [6].

4 Translating Object Propositions into Boogie

In order for us to be able to use Z3 for the verification of the generated conditions, we need to encode our extended fragment of linear logic into Boogie, which is based on first order logic and uses maps as first class citizens of the language. A detailed description of the syntax and semantics of the Boogie language can be found in [18] available at `http://research.microsoft.com/~leino/papers.html`. By 'extended fragment of linear logic' we mean the fragment of linear logic containing the operators $\otimes$ and $\oplus$, that we extend with the specifics of our object propositions methodology. Specifically, we need to encode R described in the grammar in Section 2. The crux of the encoding is in how we treat the fractions of the object propositions, how we keep track of them and how we assert statements about them. For object propositions, we encode whether they are packed or unpacked, the amount of the fraction that they have and the values of their parameters. Fractions are intrinsically related to keeping track of resources, the principal idea of linear logic. The challenge was to capture all the properties of the Oprop language and soundly translate them into first order logic statements. We were able to use the *map* data structure that the Boogie language provides to model the heap and the fields of each class. The maps were also helpful for keeping track of the fractions associated with each object, as well as for knowing which object propositions were packed and which were unpacked at all points in the code.

Our translation of linear logic (LL) into first order logic (FOL) is given in the following paragraphs of this subsection, where we present the most interesting rules of translation of our Oprop language into the Boogie intermediate verification language. The translation of SimpleCell.java from Figure 2 into the Boogie language is given in Figures 3 and 4, and we point to the lines in these two figures for each translation rule that we describe. We give the intuition for each translation rule, in the order that it appears in Figures 3 and 4.

At the start of each Boogie program we declare the type `Ref` that represents object references, as can be seen on line 1 in the `SimpleCell` example.

A class declaration is made of the field, predicate, constructor and method declarations. The function trans represents the formal translation function. For a complete list of translation rules and their explanations, please refer to Section 4.3.1 in [22].

$\text{trans}(\mathsf{ClDecl}) ::= \overline{\text{trans}(\mathsf{FldDecl})}\ \overline{\text{trans}(\mathsf{PredDecl})}$

```
type Ref;
const null: Ref;
var val: [Ref]int;
var next: [Ref]Ref;
var packedPredNext: [Ref] bool;
var fracPredNext: [Ref] real;
var packedPredVal: [Ref] bool;
var fracPredVal: [Ref] real;

procedure PackPredNext(obj: Ref, this:Ref);
  requires (packedPredNext[this]==false) &&
    (((fracPredVal[next[this]] >= 0.34))) && (next[this]==obj)
      ;
procedure UnpackPredNext(obj: Ref, this:Ref);
  requires packedPredNext[this] &&
    (fracPredNext[this] > 0.0);
  requires (next[this]==obj);
  ensures (((fracPredVal[next[this]] >= 0.34))) && (next[this
     ]==obj);

procedure PackPredVal(v:int, this:Ref);
  requires (packedPredVal[this]==false) &&
    ((v<15)) && (val[this]==v);
procedure UnpackPredVal(v:int, this:Ref);
  requires packedPredVal[this] &&
    (fracPredVal[this] > 0.0);
  requires (val[this]==v);
  ensures ((v<15)) && (val[this]==v);

procedure SimpleCell(v:int, n:Ref, this:Ref)
  modifies next,val;
  ensures ((val[this]==v)&&(next[this]==n));
  ensures (forall x: Ref::((x!=this)==>(next[x]==old(next[x]))
      ));
  ensures (forall x: Ref::((x!=this)==>(val[x]==old(val[x]))))
      ;
{ val[this]:=v;
  next[this]:=n; }

procedure changeVal(r:int, this:Ref)
  modifies packedPredVal,val;
  requires (this != null) && (((packedPredVal[this] ) &&
    (fracPredVal[this] > 0.0))&&(r<15));
  ensures ((packedPredVal[this] ) &&
    (fracPredVal[this] > 0.0));
  requires (forall x:Ref :: packedPredVal[x]);
  ensures (forall x:Ref :: packedPredVal[x]);
  ensures (forall x:Ref :: (fracPredVal[x]==old(fracPredVal[x
     ])));
```

Fig. 3. simplecell.bpl

```
{
  assume (forall y:Ref :: (fracPredVal[y] >= 0.0) );
  call UnpackPredVal(val[this], this);
  packedPredVal[this] := false;
  val[this]:=r;
  call PackPredVal(r, this);
  packedPredVal[this] := true;
}

procedure main(this:Ref)
  modifies fracPredNext,fracPredVal,next,
    packedPredNext,packedPredVal,val;
  requires (forall x:Ref :: packedPredNext[x]);
  requires (forall x:Ref :: packedPredVal[x]);
{
  var c:Ref;
  var a:Ref;
  var b:Ref;
  assume (c!=a) && (c!=b) && (a!=b) ;
  assume (forall y:Ref :: (fracPredNext[y] >= 0.0) );
  call SimpleCell(2,null,c);
  packedPredVal[c] := false;
  call PackPredVal(2,c);
  packedPredVal[c] := true;
  fracPredVal[c] := 1.0;
  call SimpleCell(2,c,a);
  packedPredNext[a] := false;
  call PackPredNext(c,a);
  fracPredVal[c] := fracPredVal[c] - 0.34;
  packedPredNext[a] := true;
  fracPredNext[a] := 1.0;
  call SimpleCell(3,c,b);
  packedPredNext[b] := false;
  call PackPredNext(c,b);
  fracPredVal[c] := fracPredVal[c] - 0.34;
  packedPredNext[b] := true;
  fracPredNext[b] := 1.0;
  call UnpackPredNext(c, a);
  fracPredVal[c] := fracPredVal[c] + 0.34;
  packedPredNext[a] := false;
  call UnpackPredNext(c, b);
  fracPredVal[c] := fracPredVal[c] + 0.34;
  packedPredNext[b] := false;
  call changeVal(4,c);
}
```

Fig. 4. simplecell.bpl - cont.

$\overline{\text{trans(ConstructorDecl)}}$ $\overline{\text{trans(MthDecl)}}$

Each field is represented by a map from object references to values, representing the value of that field. You can see the maps declared for the fields `val` and `next` on lines 3 and 4.

trans(FldDecl) ::= `var f: [Ref]`trans(T);

We declare a map from a reference `r` to a real representing the fraction `k` for each object proposition `r#k Q(`$\bar{t}$`)`. We declare a second map from a reference `r` to a boolean, keeping track of which objects are packed. Each key points to `true` if and only if the corresponding object proposition is packed for that object. For each predicate `Q`, we have a map keeping track of fractions and a map keeping track of the packed objects. The result of these translation rules is shown on lines 5 to 8 in the `SimpleCell` example.

For each predicate `Pred` we have a map `fracPred` declared as follows

`var fracPred : [Ref] real`. You can see two such maps on lines 6 and 8 of Figure 3 representing the fraction maps for the predicates `PredNext` and `PredVal`. For each object `obj`, this map points to the value of type real of the fraction `k` that is in the object proposition `obj#k Pred(`$\bar{t}$`)`. The map `fracPred` represents all the permissions on the stack. Since `fracPred` is a global variable it always contains the values of the fractions for all objects. An important distinctions is that in a method we only reason about the values stored in `fracPred` for locally accessible objects (objects that are mentioned in the precondition of that method). As we go through the body of that method, the value of `fracPred` for the objects that are touched in any way changes.

The declaration of an abstract predicate `Q` has two steps: we write a procedure *PackQ* that is used for packing the predicate `Q`, and a procedure *UnpackQ* that is used for unpacking it.

The procedure `PackQ` is called in the code whenever we have to pack an object proposition, according to the *pack(...)* annotations that the programmer inserted in the code. Right after calling the *PackQ* procedure, we write `packedQ[this] := true;` in the Boogie code. After calling the *PackQ* procedure, we also write the statements that manipulate the fractions that appear in the body of the predicate that we are packing. When packing a predicate, we subtract from the current value of fractions.

Whenever there is a packed object proposition in the body of a predicate, for example assume that in the body of the predicate `P` we have the packed object proposition `r1#k Q()`, we model it in the following way: in the procedures `UnpackP` and `PackP` we have `requires (fracQ[r1] > 0.0)` and `ensures (fracQ[r1] > 0.0)` respectively. Note that we do not have `requires packedQ[r1]` and `ensures packedQ[r1]` respectively, in the body of a predicate. The intuition here is that we do not know whether an object proposition appearing in the definition of a predicate is unpacked. For the pre- and post-conditions of procedures our methodology guarantees that if an object proposition is unpacked, it will appear in the specifications as unpacked. All unpacked object propositions will be stated in the specifications of methods. For all procedures where we might have unpacked object propositions in the pre-conditions of that procedure, we add a statement of the form `requires (forall y:Ref :: (y!=obj1) ==> packedQ[y])`. This statement states that all the object propositions that are not explicitly mentioned to be unpacked will be considered packed. For the `requires forall` statement just mentioned, assume that the object `obj1` has been explicitly mentioned in an unpacked object proposition. The upside is that we can rely on such `ensures forall` statements but we also need to prove them as post-conditions of procedures. You can see the procedures `PackPredNext` and `PackPredVal` for predicates `PredNext` and `PredVal` on lines 10 to 12 and 19 to 21 respectively.

Note that the predicate `PredVal` has an existential statement in its definition, for the variable `v` for which we instead use the global variable of the field `val` in the Boogie translation. The implementation of this idea can be seen in the `SimpleCell` example on line 21, where the `val` field is existentially quantified and the Boogie global variable `val[this]` is used instead in the definition of the predicate. We side-effect the predicate `PredVal` to add the existential parameter `v`, as can be seen in the formal translation rule from Figure 5. This parameter is added to the list of parameters of the enclosing predicate, as can be seen on line 19.

```
function addExistential(trans(R), t:trans(T))
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be methodOrPred(params, t:
        trans(T));
    result := methodOrPred(params, t:trans(T));
    return result;
}
```

Fig. 5. addExistential() translation helper function

Similarly, the procedure `UnpackQ` is called in the code whenever we need to unpack an object proposition, whenever we need to access the field of an object or we need to add together fractions in order to get the right permission (usually when we need a permission of 1 in order to modify a predicate). The procedure `UnpackQ` is inserted in the code whenever the programmer inserted the *unpack(...)* annotation in the Java code. Right after calling the procedure `UnpackQ`, we write `packedQ[this] := false;` in the code. We also write the statements that add to the fractions that appear in the body of the predicate that we are packing. You can see the procedures `UnpackPredVal` and `UnpackPredNext` for predicates `PredVal` and `PredNext` on lines 22 to 26 and 13 to 17. Note that in the `ensures` statement of the predicate `PredNext` we did not write `fracPredNext[this] >= 0.34`. This not needed because we are going to add this fraction in the caller, right after calling `UnpackPredNext`.

For each class we write a constructor. For the class `SimpleCell` the translation of the constructor is on lines 28 to 34.

A method is translated into a `procedure` in Boogie.

$$\begin{array}{ll}\text{trans}(\mathsf{MthDecl}) ::= & \texttt{procedure m(}\ \overline{\text{trans}(\mathsf{T})\ \texttt{x}}\ \texttt{) returns (}\mathsf{r}\text{:trans}(\mathsf{T})\texttt{)} \\ & \text{trans}(\mathsf{MthSpec}) \\ & \{\ \text{assume (forall y:Ref :: (fracQ[y] >= 0.0));} \\ & \overline{\text{trans}(e_1)}\ ;\ \texttt{var r :=}\ \text{trans}(e_2)\texttt{; return r;}\ \}\end{array}$$

When specifying a method, we have to specify the variables that it modifies, its precondition and its postcondition. We define the method `changeVal` in the `SimpleCell` class on lines 36 to 57. For each method we have added two kinds of statements that we call `requires forall` and `ensures forall`.

The `requires forall` statement explicitly states the object propositions that are packed at the beginning of a method, which are almost all object propositions in most cases. Since there are no unpacked object propositions in the preconditions of the method `changeVal`, the `requires forall` states that all the object propositions for the `PredVal` predicate are packed. Each `requires forall` or `ensures forall` statement refers to a single predicate and thus each method might have multiple such statements.

The `modifies` clause of a procedure in Boogie has to state all the global variables that this procedure modifies, through assignment, and all the variables that are being modified by other procedures that are called inside this procedure. The `modifies` clause that Boogie needs for each procedure states that all the values of a certain field have been modified, for all references. This leads to us not being able to rely on many properties that were true before entering a procedure. We counteract the effect of the `modifies` by adding statements of the form `ensures (forall y:Ref :: (fracP[y] == old(fracP[y]) ) )` and

`ensures (forall y:Ref :: (packedP[y] == old(packedP[y]) ) )`, for all `fracP`, `packedP` or global fields maps that were mentioned in the `modifies` clause of the current procedure. Of course, if the value of the maps `fracP`, `packedP`, etc. does change in the method we do not add these `ensures forall` statements.

If we have multiple declarations of the form `var c: Ref`, `var d: Ref`, we also add the assumption statement `assume (c!=d)` as on line 68 because the Boogie tool does not assume that these two variables are different, while the Java semantics does assume this. We explicitly assume in the beginning of the body of each method that all fractions that are mentioned in the pre- or post-conditions of that method are larger than 0, as on line 69. We need to explicitly add these assumptions because the Boogie tool does not have any pre-existing assumptions about our global maps representing fractions.

An object proposition `r#k Q(`$\bar{t}$`)` is generally translated by stating that the value of the `packedQ` map for the parameters `t` and reference `r` is true and the value of `fracQ` for the same parameters and reference is $>= k$ if k is a constant or is > 0 if k is a metavariable. You can see the translation of the packed object proposition `obj#0.34 PredVal()` both inside the `PackPredNext` and `UnpackPredNext` procedures corresponding to the predicate `PredNext`.

When the programmer wants to pack an object to a predicate `Q`, he needs to write the statement `call PackQ(..., this)` in the program. When translating this call to `Pack`, the Oprop tool writes `packedQ[this] := true`. You can see an example of such a call to `Pack` on lines 71 to 74, together with the statements that assign `true` to the global `packed` map for this object and the statement that subtracts the fraction that is used for the predicate `PredVal` and the object `c` when the packing occurs. The user specifies to our tool which predicate they intend to pack to by adding the name of that predicate and the parameters to that predicate, if there are any, in the call to the constructor. This can be seen on lines 17-19 in Figure 2 right after the name of the constructor `SimpleCell`. For our `SimpleCell` example the constructor is called multiple times in our `main` function and you can see one such call and the statements that are written right after the call on lines 70 to 74 in the `.bpl` translation. The user specifies to our Oprop tool which predicate they intend to pack to by writing the name of the predicate, together with any parameters that the predicate needs, in the call to the constructor.

Similarly, when we unpack an object from an object proposition that refers to predicate `Q`, we write the statement `call UnpackQ(..., this)`. Right after this statement we write `packedQ[this] := false`. You can see an example of such a call on lines 87 to 89. As opposed to packing, where we usually consume a fraction of an object proposition, when we unpack an object proposition we obtain a fraction and so we have a fraction manipulation statement that adds to the current value of a fraction, as seen on line 88.

5 Equivalence of Translation

In this section, we present soundness and completeness theorems stating that a formula in the Oprop language is translated into an equivalent formula in the first-order logic supported by Boogie. We focus on the equivalence of formulas because that is the heart of our translation approach; modeling the detailed constructs of Java has been done in other settings and is of less interest. Furthermore, the semantics of Boogie are given as trace sets, a formalism somewhat distant from the standard dynamic semantics we used in our prior work. Regardless, an informal argument for the equivalence of the rest of the translation is available in the first author's thesis [22] for the interested reader.

We present semantics for our subset of linear logic in Figures 6 and 7. In these figures Γ contains typings for term variables, Π_0 contains the persistent truths and Π_1 contains the resources. We have adapted these rules from a particularly elegant formalization of linear logic from Prof. Frank Pfenning's 2012 Carnegie Mellon course notes [25]. Note that we have divided the Π that we used in Section 2 into Π_0 and Π_1, to separate the persistent truths and the ephemeral resources. In fact the context Π contains the preconditions of the particular method inside which we have to prove a formula R and we have the equality $\Pi = \Pi_0; \Pi_1$. We are presenting the entailment relation in this section to both define precisely what it means in our system and allow us to prove properties about it.

In our restricted setting, the main difference between linear and classical logic is that object propositions are treated as resources, with the amount of

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}}) \otimes r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}})}{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k}\ Q(\bar{\mathsf{t}})}\ (OPack_1)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k*2}\ Q(\bar{\mathsf{t}})}{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k}\ Q(\bar{\mathsf{t}}) \otimes r\#\mathsf{k}\ Q(\bar{\mathsf{t}})}\ (OPack_2)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}}) \otimes \mathtt{unpacked}(r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}}))}{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathtt{unpacked}(r\#\mathsf{k}\ Q(\bar{\mathsf{t}}))}\ (OUnpack_1)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathtt{unpacked}(r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}})) \otimes \mathtt{unpacked}(r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}}))}{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathtt{unpacked}(r\#\mathsf{k}\ Q(\bar{\mathsf{t}}))}\ (OUnpack_2)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathtt{unpacked}(r\#\mathsf{k*2}\ Q(\bar{\mathsf{t}}))}{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathtt{unpacked}(r\#\mathsf{k}\ Q(\bar{\mathsf{t}})) \otimes \mathtt{unpacked}(r\#\mathsf{k}\ Q(\bar{\mathsf{t}}))}\ (OUnpack_3)$$

$$\frac{\Gamma;\Pi_0;\Pi_1, r.f \to x \vdash\ true}{\Gamma;\Pi_0;\Pi_1 \vdash\ r.f \to \mathsf{x}}\ (OField)$$

$$\frac{\Gamma \vdash\ \mathsf{t1} : T \quad \Gamma \vdash\ \mathsf{t2} : T \quad \Gamma \vdash\ \mathsf{binop} : (T, T) \to T_2 \quad \Gamma;\Pi_0, \mathsf{t1\ binop\ t2};\Pi_1 \vdash\ true}{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathsf{t1\ binop\ t2}}\ (OBin)$$

Fig. 6. Semantics for Linear Logic Formulas

resource represented by the fraction. A threat to sound translation is that a partial fraction of a proposition in the context could be used twice in the classical setting, but must only be used once in the linear setting. To prevent this, our translation converts the formula to disjunctive normal form and coalesces all fractions within each disjunction, thus ensuring that the entire fraction required is present in the context even in the classical setting.

Theorem 1 (Completeness Theorem) *For a formula R that is written in linear logic and parses according to the grammar in Section 2, if $\Gamma;\Pi_0;\Pi_1 \vdash^{LL} R$ then trans$(\Gamma;\Pi_0;\Pi_1) \vdash^{FOL}$ trans(R).*

Proof. The proof is by induction on the rules in Figures 6 and 7. For each formula R we have to prove that if R is true in linear logic then trans(R) is true in first order logic. Due to space constraints, we show a representative case and leave the rest to [22], Section 4.5.

– **Case** $OPack_1$
  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ P then trans$(\Gamma;\Pi_0;\Pi_1) \vdash^{FOL}$ trans(P) where:
  1. P $= r\#\mathsf{k}\ Q(\bar{\mathsf{t}})$
  2. trans(P) = `packedQ[r]` && translateObjectProposition(`r#k Q(`$\bar{t}$`)`) and
  3. $$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}}) \otimes r\#\mathsf{k/2}\ Q(\bar{\mathsf{t}})}{\Gamma;\Pi_0;\Pi_1 \vdash\ r\#\mathsf{k}\ Q(\bar{\mathsf{t}})}\ (OPack_1)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R1} \quad \Gamma;\Pi_0;\Pi_1' \vdash \mathsf{R2}}{\Gamma;\Pi_0;\Pi_1,\Pi_1' \vdash \mathsf{R1} \otimes \mathsf{R2}}\ (\otimes)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R1}}{\Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R1} \oplus \mathsf{R2}}\ (\oplus_L)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R2}}{\Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R1} \oplus \mathsf{R2}}\ (\oplus_R)$$

$$\frac{\Gamma \vdash \mathsf{M{:}T} \quad \Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R\{M/x\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \exists \mathsf{x{:}T.R}}\ (\exists_1)$$

$$\frac{\Gamma \vdash \mathsf{F{:}double} \quad \Pi_0 \vdash \mathsf{F} > 0 \quad \Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R\{F/z\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \exists \mathsf{z{:}double.R}}\ (\exists_2)$$

$$\frac{\Gamma \vdash \mathsf{F{:}double} \quad \Pi_0 \vdash \mathsf{F\ binop\ t} \quad \Gamma;\Pi_0;\Pi_1 \vdash \mathsf{R\{F/z\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \exists \mathsf{z{:}double.z\ binop\ t} \Rightarrow \mathsf{R}}\ (\exists_3)$$

$$\frac{\Gamma, m:T;\Pi_0;\Pi_1 \vdash \mathsf{R\{m/x\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \forall \mathsf{x{:}T.R}}\ (\forall_1)$$

$$\frac{\Gamma, \mathsf{f{:}double};\Pi_0, \mathsf{f} > 0;\Pi_1 \vdash \mathsf{R\{f/z\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \forall \mathsf{z{:}double.R}}\ (\forall_2)$$

$$\frac{\Gamma, f : \mathit{double};\Pi_0, \mathsf{f\ binop\ t};\Pi_1 \vdash \mathsf{R\{f/z\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \forall \mathsf{z{:}double.z\ binop\ t} \Rightarrow R}\ (\forall_3)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash \mathsf{t1\ binop\ t2} \Rightarrow \mathsf{R}}{\Gamma;\Pi_0, \mathsf{t1\ binop\ t2};\Pi_1 \vdash \mathsf{R}}\ (\mathit{tbint})$$

$$\frac{}{\Gamma;\Pi_0;\Pi_1, \mathsf{R} \vdash \mathsf{R}}\ (\mathit{id})$$

Fig. 7. Semantics for Linear Logic Formulas - cont.

```
function translateObjectProposition(r#k Q(t))
returns String {
    String result = "";
    if (k is a constant) {
        result += "(fracQ[r] >= k) &&";
    } else {
        result += "(fracQ[r] > 0.0) &&"; }
    if parameter t corresponds to a field of r {
        say that field is field1;
        result += "(field1[r]==t)"; }
    result += body of predicate Q with the formal parameters
              replaced by the actual ones;
    return result;
}
```

Fig. 8. translateObjectProposition() translation helper function

Proof:
From the $OPack_1$ rule we know that
if $\Gamma; \Pi_0; \Pi_1 \vdash^{LL} r\#\mathsf{k}\ Q(\bar{\mathsf{t}})$ then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL} r\#\mathsf{k}/2\ Q(\bar{\mathsf{t}}) \otimes r\#\mathsf{k}/2\ Q(\bar{\mathsf{t}})$
. Using the induction hypothesis, we know that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans($\vdash^{LL}$ $r\#\mathsf{k}/2\ Q(\bar{\mathsf{t}}) \otimes r\#\mathsf{k}/2\ Q(\bar{\mathsf{t}})$). Using the definition of the translateAnd() function from Figure 9, we obtains that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans($r\#(\mathsf{k}/2 + \mathsf{k}/2)\ Q(\bar{\mathsf{t}})$), i.e., trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans($r\#(\mathsf{k})\ Q(\bar{\mathsf{t}})$).
This means that if P holds in LL then trans(P) holds in FOL. Q.E.D.

Theorem 2 (Soundness Theorem) *For a formula R that is written in linear logic and parses according to the grammar in Section 2, if trans*($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ *trans(R) then* $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ *R.*

Proof. The proof will follow the cases of R in the grammar in Section 2, but it will be done by induction on the complexity of formula R. For each formula R we have to prove that if trans(R) is true in first order logic then R is true in linear logic.

– **Case** $P3$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(P) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ P, where:
  1. $\mathsf{R} = \mathsf{P} = r.f \to \mathsf{x}$
  2. trans(P) = $(f[r] == x)$

  Proof:
  If $(f[r] == x)$ holds in FOL, by the identity rule from FOL we know that $(f[r] == x)$ is in trans(Π_1). Knowing that we devised f to be a map that for the key r holds the current value of field $r.f$, it means that the value of field f of reference r is equal to x in LL.

The other cases of the formal proof can be found in [22], Section 4.5. In section 4.6 of [22] we also present an informal soundness argument.

```
function translateAnd(R1, R2) returns FOLFormula {
  let R = DNF(R1 cross R2)
  let R' = FOL(coalesce(R))
  return transAtoms(R') }
where
  DNF(R) converts linear formula R to disjuctive normal form
  coalesce(R) merges atoms in the same conjunction by adding
      fractions
  FOL(R) replaces linear connectives with first-order logic
      connectives
  transAtoms(R) translates the atoms of R, leaving connectives
       unchanged
}
```
Fig. 9. translateAnd() translation helper function

6 Evaluation

The Composite design pattern [10] 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 [15], 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 proposed by Leavens *et al.* [15], with some attempted solutions presented at SAVCBS 2008 (e.g. [4, 14]). We have already presented our formalization and manual proof of the Composite pattern using fractions and object propositions in our published paper [23]. One of the biggest accomplishments that we present in this paper is that we were able to automatically verify the Composite pattern using the Oprop tool. The fact that our tool can automatically verify the challenging Composite pattern is proof of its maturity.

The annotated Composite.java file be seen in the example.zip folder that can be downloaded from the first page of the Oprop web application at *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com.* The Composite.java file presents the predicates `left, right, count` and `parent`, together with the annotated methods `updateCountRec, updateCount` and `setLeft` that we are able to modularly verify independently of each other, just by looking at their pre- and post-conditions. 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. 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 add a new subtree at any time, to any free position. This operation changes the count of all ancestors, which is done through a notification protocol. The pattern of circular dependencies and the notification mechanism are hard to capture with verification approaches based on ownership or uniqueness.

This folder also contains the SimpleCell.java file, together with the files DoubleCount.java and Link.java that we formally verified using Oprop. The class DoubleCount.java represents objects which have a field *val* and a field *dbl*, such that *dbl==2*val.* This property represents the invariant of objects of type Doublecount. We want to verify that this invariant is maintained by the method *increment.* The example Link.java illustrates how we deal with predicates that have parameters. As we verify more programs, we are adding examples to this folder. Note that when an example cannot be verified, the user will see a list of errors produced by the Boogie backend, detailing which specifications could not be proved. In that case the user should go back to the original example and modify the pre- and post-conditions of methods, or the pack/unpack annotations in the body of methods so that the verification can be successfully performed.

7 Related Work

Other researchers have encoded linear logic or fragments of it into first order logic. In his Ph.D. thesis [26] Jason Reed presents an encoding of the entirety of linear logic into first order logic. The major technical difference between Reed's encoding and ours is that he encodes uninterpreted symbols while our encoding is done inside the theory of object propositions - the smaller fragment of linear logic on top of which we have added the object propositions. His encoding is suited to any formula written in linear logic, irrespective of its meaning, while ours is targeted towards formulas written in our extended fragment of linear logic, that have a specific semantics.

Heule *et al.* [12] present an encoding of abstract predicates and abstraction functions in the verification condition generator Boogie. Their encoding is sound and handles recursion in a way that is suitable for automatic verification using SMT solvers. It is implemented in the automatic verifier Chalice. Since our system differs from theirs in the way we handle fractions (they need a full permission in order to be able to modify a field, while we are able to modify fields even if we have a fractional permission to the object enclosing the field), we came up with an encoding that is specific to our needs in our Oprop tool.

Peter Müller *et al.* [19] created the Viper toolchain, that also uses Boogie and Z3 as a backend, and can reason about persistent mutable state, about method permissions or ownership, but they also need a full permission to modify shared data. There are other formal verification tools for object oriented programs, such as KeY [1], VCC [6] or Dafny [17], that implement other methodologies.

8 Conclusion

We have presented the Oprop tool that implements the object proposition methodology, a modular approach that can be used successfully in the verification of component-based software. We described the translation rules on which the tool is based, by referring to one example class and we proved the equivalence between formulas in Oprop and their translation into Boogie. We gave insight into the automatic verification of an instance of the Composite pattern and other examples and described the mode of usage of our web application where the tool can be accessed.

References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
2. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.
3. 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.
4. Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the composite design pattern. In *Proc of SAVCBS 2008*, 2008.

5. John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003.
6. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.
7. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. ETAPS, pages 337–340, Berlin, Heidelberg, 2008.
8. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
9. Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
11. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.
12. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP*, 2013.
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. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, June 2007.
16. K. Rustan Leino and Peter Muller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
17. K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
18. K.R.M. Leino. This is boogie 2. Manuscript KRML 178. 2008.
19. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.
20. Peter Müller. Modular specification and verification of object-oriented programs. In *Lecture Notes in Computer Science*, volume 2262, 2002.
21. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. In *Science of Computer Programming*, volume 62(3), pages 253–286, 2006.
22. Ligia Nistor. CMU Ph.D. thesis (in preparation), `http://www.cs.cmu.edu/~lnistor/thesis.pdf`.
23. Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. Object propositions. In *FM*, 2014.
24. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
25. Frank Pfenning. `http://www.cs.cmu.edu/~fp/courses/15816-s12/`. 2012.
26. Jason Reed. A hybrid logical framework. In *PhD thesis. Technical Report CMU-CS-09-155*, 2009.
27. 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.