The Oprop Verification Tool: Object Propositions in Action

Ligia Nistor
School of Computer Science, Carnegie Mellon University
lnistor@cs.cmu.edu

Abstract

We have recently introduced object propositions as a modular verification technique that combines abstract predicates and fractional permissions. The Oprop tool implements the theory of object propositions and 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. We present the details of our implementation, the lessons that we learned and a number of examples that we have verified using the Oprop tool.

1. Motivation

The formal verification of object oriented code is still an open problem. The presence of aliasing makes modular verification difficult: if there are multiple clients depending on the properties of an object, one client may break the property that others depend on. Existing verification approaches use the classical invariant technique [8] or separation logic [14] but they have limitations such as all objects of the same class needing to satisfy the same invariant or the specification of methods needing to reveal the exact structure of the objects that they modify.

Object propositions bypass these limitations and they are a step forward in the modular verification of object oriented code. They 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.

2. The Theory of Object Propositions

The object proposition methodology [13] uses abstract predicates [14] to characterise the state of an object, embeds those predicates in a logical framework, and specifies sharing using fractional permissions [9]. 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 [10], our verification system maintains a *key* (represented $o.f \rightarrow x$) for each field of the receiver object, which is used to track the current values of those fields through the method.


```
class DoubleCount {
  int val; int dbl;
  predicate OK() =
    exists int v, int d :
    this.val -> v && this.dbl -> d && d == 2*v
  void increment()
  ~double k:
  requires this#k OK()
  ensures this#k OK()
  { unpack(this#k OK());
   this.val = this.val + 1;
   this.dbl = this.dbl + 2;
   pack(this#k OK()); } }
```

Figure 1. DoubleCount.java

The programming language that we are using is inspired by Featherweight Java [11], extended to include object propositions. Our tool, Oprop, is unique because it implements the theory of object propositions. There are other formal verification tools for object oriented programs, such as KeY [6] or Dafny [2], but they implement other methodologies. We have formal translation rules from the Oprop language into the Boogie intermediate verification language.

3. Example

The class DoubleCount.java is given in given in Figure 1. The DoubleCount example has been previously briefly covered in the AI4FM'14 workshop [12]. There we only presented a sketch of a manual translation while here the example is translated automatically by our Oprop tool. The class DoubleCount 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 tilde sign in the specification of the `increment` method in Figure 1 is there to differentiate between variables `k` used for fractions and other variables that are used as parameters to predicates.

4. Translating the Oprop Language into Boogie

The Boogie tool [1] uses the first order logic theorem prover Z3 [4] for the verification of input programs. This means that we need to encode our extended fragment of linear logic representing Oprop into first order logic. The crux of the encoding is in how we treat fractions, how we keep track of them and how we assert statements about them. Fractions are intrinsically related to keeping track of resources, the main idea of linear logic. We are working on proving that our Boogie translation and Oprop are semantically equivalent.

At the start of each Boogie program we declare the type `Ref` that represents object references. We declare a map from the parameters of a predicate and a reference `r` to a real in the interval [0,1]

representing the fraction k of the object proposition r@k Q($\bar{t}$). We declare a second map from the parameters of a predicate (Q for the above object proposition) and 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 each predicate Q, we have a map keeping track of fractions and a map keeping track of the packed objects.

trans(Prog) ::= type Ref;
var fracQ : [Ref] real;
var packedQ : [Ref] bool;
const null : Ref;
trans($\overline{\text{C|Dec|}}$) trans(e)

A class declaration is made of the field, predicate, constructor and method declarations. Because of lack of space, we stop here in presenting the translation rules.

5. The Oprop Tool

The Oprop tool [5] takes as input any number of files written in Java and annotated with object propositions specifications. For each file, it produces the corresponding Boogie translation file. If the user has provided multiple files as input, there will be multiple files produced as output. The user will have to manually concatenate the multiple files into a single one. The user will then go to the webpage *http://rise4fun.com/boogie*, that is the web interface to the Boogie verification system, to find out whether 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.

The Oprop tool is composed of two parts: JExpr [7] and the Boogie translation. JExpr is a parser for a very small subset of Java, developed by Paul Cager. We took his 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 found in the file *JExpr.jj*, 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 our formal translation rules. By implementig the visitor design pattern, we visit all the nodes of the Abstract Syntax Tree and perform the translation for each one.

6. Translation of Example

In this section we describe how the translation rules are applied in practice. The Boogie translation of DoubleCount.java is given in Figure 2. We have global map variables for the fields of the class, for keeping track of which objects are packed and for the fraction corresponding to each object proposition. We have the translation of the constructor ConstructDoubleCount, procedures PackOK and UnpackOK that are being called when an object has to be packed or unpacked. In the specification of procedure increment, we require that all objects are packed on the entrance to the procedure. In our methodology, all the objects that are not explicitly unpacked are considered packed. The ensures forall and requires forall specifications act as frame conditions and restrict the number of objects that the Boogie verifier assumes have changed at method boundaries. We need this restriction because of the modifies that Boogie needs for each method: the modifies is very general and assumes that all the global variables have been changed, thus nullifying any previous (old) properties about those variables.

Examples SimpleCell.java, Link.java and Share.java can be seen in our tool directory [5]. The current version of the code behind Oprop can be found on GitHub [3]. We created the SimpleCell example to illustrate how we add fractions in the cases when we need a larger fraction to an object proposition than we currently have. The example Link illustrates how we deal with predicates that have parameters. We created the Share example to exemplify objects that have a reference to a *shared* common object.

```
type Ref;
const null: Ref;
var val: [Ref]int;
var dbl: [Ref]int;
var packedOK: [Ref] bool;
var fracOK: [Ref] real;
procedure ConstructDoubleCount
    (val1 :int, dbl1 :int, this: Ref);
  ensures (val[this] == val1) && (dbl[this] == dbl1);
procedure PackOK(this:Ref);
  requires packedOK[this]==false && ((dbl[this]==(2*val[this])));
procedure UnpackOK(this:Ref);
  requires packedOK[this] &&(fracOK[this] > 0.0);
  ensures ((dbl[this]==(2*val[this])));
procedure increment(this:Ref)
  modifies dbl,packedOK,val;
  requires packedOK[this] && (fracOK[this] > 0.0);
  ensures packedOK[this] && (fracOK[this] > 0.0);
  requires (forall x:Ref :: packedOK[x]);
  ensures (forall x:Ref::(packedOK[x] == old(packedOK[x])));
    { call UnpackOK(this);
    packedOK[this] := false;
    val[this]:=val[this]+1;
    dbl[this]:=dbl[this]+2;
    call PackOK(this);
    packedOK[this] := true;}
```

Figure 2. doublecount.bpl

7. Aknowledgements

We thank Prof. Jonathan Aldrich, who is Ligia Nistor's doctoral advisor, for his guidance. Jihyun Lee (*jhlee248@gmail.com*), who was a Master's student at Carnegie Mellon University, also contributed to the implementation of Oprop.

References

[1] http://rise4fun.com/boogie.

[2] http://rise4fun.com/dafny.

[3] https://github.com/ligianistor/Oprop.

[4] https://github.com/Z3Prover/z3.

[5] http://www.cs.cmu.edu/~lnistor/src15.zip.

[6] http://www.key-project.org.

[7] http://www.paulcager.org/products/minijava/.

[8] 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.

[9] John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003.

[10] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

[11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *TOPLAS*, pages 132–146, 2001.

[12] Ligia Nistor and Jonathan Aldrich. Using machine learning in the automatic translation of object propositions. In *AI4FM*, 2014.

[13] Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. Object propositions. In *FM*, 2014.

[14] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.