Thesis Proposal: Object Propositions

Ligia Nistor

Computer Science Department, Carnegie Mellon University
lnistor@cs.cmu.edu

1 Introduction

The formal verification of object-oriented code is still an open problem. The presence of aliasing (multiple references pointing to the same object) makes modular verification of code difficult. If there are multiple clients depending on the properties of an object, one client may break the property that others depend on. In order to verify whether clients and implementations are compliant with specifications, knowledge of both aliasing and properties about objects is needed.

Statically verifying the correctness of code is bound to save time in the quality assurance process. Bugs are going to be discovered before the program even starts running. Formal verification saves the programmer time by helping him/her find the errors more quickly. By using an off-the-shelf formal verification tool instead of generating as many test cases as possible, the quality assurance process becomes more efficient.

Creating verification tools for object-oriented programming languages is a worthy endeavor. According to Oracle [5], the Java platform has attracted more than 6.5 million software developers to date. The Java programming language is used in every major industry segment and has a presence in a wide range of devices, computers, and networks. There are 1.1 billion desktops running Java, 930 million Java Runtime Environment downloads each year, 3 billion mobile phones running Java, 100% of all Blu-ray players run Java and 1.4 billion Java Cards are manufactured each year. Java also powers set-top boxes, printers, Web cams, games, car navigation systems, lottery terminals, medical devices, parking payment stations, and more.

We acknowledge that the world is transitioning from single-CPU machines and single-threaded programs to multi-core machines and concurrent programs. Still, in order to apply a verification procedure in a multi-threaded setting, we first have to know how to apply it for a single thread. Moreover, according to Amdahl's law [6], the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Special attention needs to be given to the sequential fragment of a program because that is where the bottleneck will be at execution time. There is a large number of applications that will not benefit from parallelism, such as any algorithm with a large number of serial steps. These algorithms will need to be implemented sequentially and will need a verification procedure designed especially for them. This thesis describes how we can use information about aliasing and properties of objects to implement a robust verification tool for object-oriented code in a single-threaded setting.

1.1 Current Approaches

The verification of object-oriented code can be achieved using the classical invariant-based technique [8]. 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, i.e. before a method is called on the object and after the method returns. The invariant can be broken inside the method as long as it is restored upon exit from the method. This leads to a key limitation: the specifications that can be written using the proof language and the verification that can be performed are limited. This limitation shows itself in a number of ways. One sign is: the 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 post-condition. Another sign is that the invariant of an object cannot depend on another object's state, unless additional features such as ownership are added. Leino and M*ü*ller [37] have added ownership to organize objects into contexts. In their approach using object invariants, the invariant of an object is allowed to depend on the fields of the object, on the fields of all objects in transitively-owned contexts, and on fields of objects reachable via given sequences of fields. A related restriction is that from outside the object, one cannot make an assertion about that object's state, other than that its invariant holds. 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 [34], [17], [15], etc. bypass the limitations of invariant-based verification techniques by requiring that each method describe its footprint and the predicates that should hold for the objects in that footprint. In this way not all objects of the same class have to satisfy the same predicate. 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. This is not a problem if the objects in the footprint are completely encapsulated. But if they are shared between two structures, that sharing must be revealed when transitioning between the "inside" and "outside" of the encapsulating abstraction. This is not desirable from an information hiding point of view.

On the other hand, permission-based work [10], [16], [13] 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. This is different from the traditional thinking that an object invariant is always the same for all objects. What share and/or fractions do is allow us to make different assertions about different objects; we are not limited to a single object invariant. This relaxes the classical invariant-based verification technique and it makes it much more flexible.

Moreover, developers can use *access permissions* [10] to express the design intent of their protocols in annotations on methods and classes. This thesis uses fractional permissions [13] (which are similar to access permissions in some respects) in the creation of the novel concept of object propositions. The main difference between the way I use permissions and existing work about permissions is that I do not require the state referred to by a fraction less than 1 to be immutable. Instead, that state has to satisfy an invariant that can be relied on by other objects. My goal is to modularly check that implementations follow their design intent. The typestate [16] formulation has certain limits of expressiveness: it is only suited to finite state abstractions. This makes it unsuitable for describing fields that contain integers (which can take an infinite number of values) and can satisfy various arithmetical properties. My object propositions have the advantage that they can express predicates over an infinite domain, such as the integers.

My fractional permissions system allows verification using a mixture of linear and nonlinear reasoning, combining ideas from previous systems. The existing work on separation logic is an example of linear reasoning, while the work on fractional permissions is an example of nonlinear reasoning. In a linear system there can be only one assertion about each piece of state (such as each field of an object), while in a nonlinear system there can be multiple mentions about the same piece of state inside a formula. The combination of ideas from these two distinct areas allows our system to provide more modularity than each individual approach. For example, in some cases our work can be more modular than separation logic approaches because it can more effectively hide the exact aliasing relationships.

The seminal work of Parnas [35] describes the importance of modular programming, where the information hiding criteria is used to divide the system into modules. In a world where software systems have to be continually changed in order to satisfy new (client) requirements, the software engineering principle of modular programming becomes crucial: it brings flexibility by allowing changes to one module without modifying the others. The examples in sections 4.1 and 1.3 represent instances where object propositions are better at hiding shared data and enforcing modularity than separation logic.

1.2 Example: Cells in a spreadsheet

I consider the example of a spreadsheet, as described in [27]. In my spreadsheet each cell contains an add formula that adds two integer inputs. Each cell may refer to other two cells. The general case would be for each cell to have a dependency list of cells, but since my grammar does not support arrays yet, I am not considering that case. Whenever the user changes a cell, each of the two cells which transitively depend upon it must be updated.

A visual representation of this example is presented in Figure 1. In separation logic, the specification of any method has to describe the entire footprint of the method, i.e., all heap locations that are being touched through reading or writing

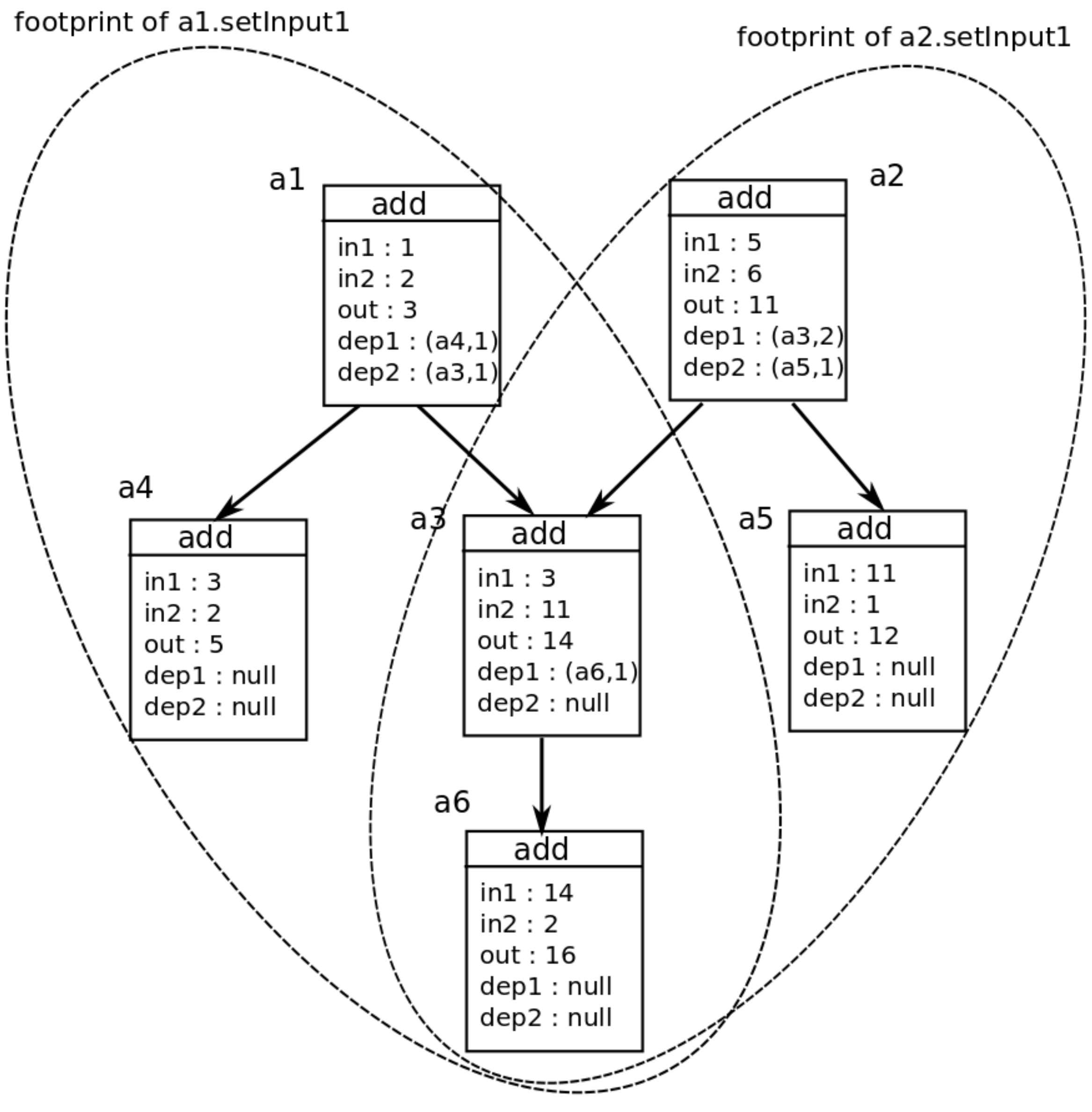

Fig. 1. Add cells in spreadsheet

in the body of the method. That is, the shared cells $a3$ and $a6$ have to be specified in the specification of all methods that modify the cells $a1$ and $a2$.

In Figure 8, I present the code implementing a cell in a spreadsheet.

```
class Dependency {
      Cell ce;
      int input;
 }
class Cell {
      int in1, in2, out;
      Dependency dep1, dep2;

  void setInputDep(int newInput) {
    if (dep1!=null) {
          if (dep1.input == 1) dep1.ce.setInput1(newInput);
          else dep1.ce.setInput2(newInput);
    }
    if (dep2!=null) {
          if (dep2.input == 1) dep2.ce.setInput1(newInput);
          else dep2.ce.setInput2(newInput);
    }
    }

  void setInput1(int x) {
    this.in1 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
    }

  void setInput2(int x) {
    this.in2 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
    }

 }
```

Fig. 2. Cell class

The specification in separation logic is unable to hide shared data. To express the fact that all cells are in a consistent state where the dependencies are respected and the sum of the inputs is equal to the output for each cell, I define the following predicate :

$SepOK(cell) \equiv (cell.in1 \to x1) \star (cell.in2 \to x2) \star (cell.out \to o) \star (cell.dep \to d) \star (x1 + x2 = o) \star \forall (c, inp) \in d : (SepOK(c) \wedge c.\text{“}in + inp\text{”} \to o)$.

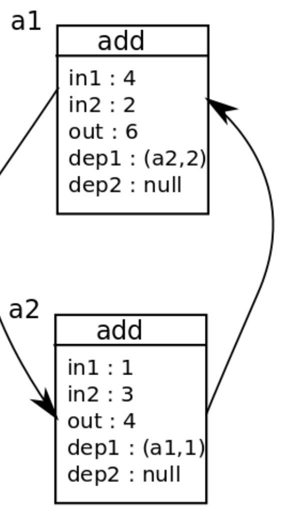

Fig. 3. Cells in a cycle

This predicate states that the sum of the two inputs of *cell* is equal to the output, and that the predicate *SepOK* is verified by all the cells that directly depend on the output of the current cell. Additionally, the predicate *SepOK* also checks that the corresponding input for each of the two dependency cells is equal to the output of the current cell. This predicate only works in the case when the cells form a directed acyclic graph (DAG). The predicate *SepOK* causes problems when there is a diamond structure (not shown in Figure 1) or if one wants to assert the predicate about two separate nodes whose subtrees overlap due to a DAG structure (e.g. a1 and a2 in Figure 1). For example, if the dependencies between the cells form a cycle, as in Figure 3, the predicate *SepOK* cannot possibly hold.

Additionally we need another predicate to express simple properties about the cells:

$Basic(cell) \equiv \exists x1, x2, o, d.(cell.in1 \rightarrow x1) \star (cell.in2 \rightarrow x2) \star (cell.out \rightarrow o) \star (cell.dep \rightarrow d)$.

Below I show a fragment of client code and its verification using separation logic.

$\{Basic(a2) \star Basic(a5) \star SepOK(a1)\}$
```
a1.setInput1(10);
```
$\{Basic(a2) \star Basic(a5) \star SepOK(a1)\}$
$\{* * * * * * * *\ missing\ step * * * * * * * * **\}$
$\{Basic(a4) \star Basic(a1) \star SepOK(a2)\}$
```
a2.setInput1(20);
```

In the specification above,

$$SepOK(a1) \equiv a1.in1 \rightarrow x1 \star a1.in2 \rightarrow x2 \star a1.out \rightarrow o \star x1 + x2 = o \star$$
$$(SepOK(a4) \wedge a4.in1 = o) \star (SepOK(a3) \wedge a3.in1 = o)$$

and

$$SepOK(a2) \equiv a2.in1 \rightarrow z1 \star a2.in2 \rightarrow z2 \star a2.out \rightarrow p \star z1 + z2 = p \star$$
$$(SepOK(a3) \wedge a3.in2 = p) \star (SepOK(a5) \wedge a5.in1 = p)$$

In separation logic, the natural pre- and post-conditions of the method *setInput1* are $SepOK(this)$, i.e., the method takes in a cell that is in a consistent state in the spreadsheet and returns a cell with the input changed, but that is still in a consistent state in the spreadsheet. The natural specification of *setInput1* would be $SepOK(this) \Rightarrow SepOK(this)$.

Thus, before calling `setInput1` on $a2$, we have to combine $SepOK(a3) \star SepOK(a5)$ into $SepOK(a2)$. We observe the following problem: in order to call `setInput1` on $a2$, we have to take out $SepOK(a3)$ and combine it with $SepOK(a5)$, to obtain $SepOK(a2)$. But the specification of the method does not allow it, hence the missing step in the verification above. The specification of `setInput1` has to be modified instead, by mentioning that there exists some cell $a3$ that satisfies $SepOK(a3)$ that we pass in and which gets passed back out again. The specification of *setInput1* would become

$\forall \alpha, \beta, x \,.\, (SepOK(this) \wedge \textit{such that } SepOK(this) \equiv \alpha \star SepOK(x) \star \beta)$
$\Rightarrow (SepOK(this) \wedge \textit{such that } SepOK(this) \equiv \alpha \star SepOK(x) \star \beta)$.

The modification is unnatural: the specification of `setInput1` should not care about which are the dependencies of the current cell, it should only care that it modified the current cell.

This situation is very problematic because the specification of `setInput1` involving shared cells becomes awkward. One can imagine an even more complicated example, where there are multiple shared cells that need to be passed in and out of different calls to `setInput1`. It is impossible to know, at the time when we write the specification of a method, on what kind of shared data that method will be used. Separation logic approaches will thus have a difficult time trying to verify this kind of code. This is because in OO design, the natural abstraction is that each cell updates its dependents, while they are hidden from the outside. The cells in the spreadsheet example is an instance of the subject-observer pattern, as described in [25], which implements this abstraction.

1.3 Modularity Example: Simulator for Queues of Jobs

The formal verification of modules should ideally follow the following principle: the specification and verification of one method should not depend on details that are private to the implementation of another method. An important instance of this principle comes in the presence of aliasing: if two methods share an object, yet their specification is not affected by this sharing, then the specification should not reveal the presence of the sharing.

Unfortunately, the most modular reasoning techniques available today — principally those based on separation logic [36] — cannot hide sharing, because the specification of a method must mention the entire memory footprint that the method accesses. This gives rise to unmodular, and therefore verbose and fragile, specifications and proofs. There exist versions of higher-order separation logics that can hide the presence of sharing to some extent [27], but their higher-order nature makes them considerably more complicated.

To illustrate the modularity issues, I present here a relatively realistic example. Figure 4 depicts a simulator for two queues of jobs, containing large jobs (size>10) and small jobs (size<11). The example is relevant in queueing theory, where an optimal scheduling policy might separate the jobs in two queues, according to some criteria. The role of the control is to make each producer/-consumer periodically take a step in the simulation. I have modeled two FIFO queues, two producers, two consumers and a control object. Each producer needs a pointer to the end of each queue, for adding a new job, and a pointer to the start of each queue, for initializing the start of the queue in case it becomes empty. Each consumer has a pointer to the start of one queue because it consumes the element that was introduced first in that queue. The control has a pointer to each producer and to each consumer. The queues are shared by the producers and consumers.

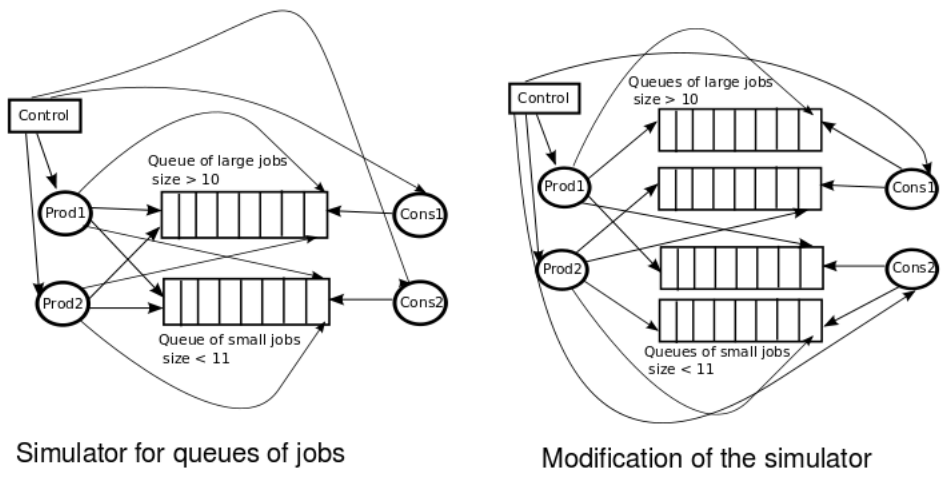
Control
Prod1
Prod2
Cons1
Cons2
Queue of large jobs size > 10
Queue of small jobs size < 11
Simulator for queues of jobs
Queues of large jobs size > 10
Queues of small jobs size < 11
Modification of the simulator

Fig. 4.

Now, let's say the system has to be modified, by introducing two queues for the small jobs and two queues for the large jobs, see right image of Figure 4. Ideally, the specification of the control object should not change, since the consumers and the producers have the same behavior as before: each producer produces both large and small jobs and each consumer accesses only one kind of job. I will show in this thesis that my methodology does not modify the specification of the control object, thus allowing one to make changes locally without influencing other code, while (first-order) separation logic approaches [18] will modify the specification of the controller.

The code in Figures 5, 6 and 7 represents the example from Figure 4.

Now, let's imagine changing the code to reflect the modifications in the right image of Figure 4. The current separation logic approaches do not provide enough modularity. Distefano and Parkinson [18] introduced jStar, an automatic verification tool based on separation logic aiming at programs written in Java. Although they are able to verify various design patterns and they can define abstract predicates that hide the name of the fields, they do not have a way of hiding the aliasing. In all cases, they reveal which references point to the same shared data, and this violates the information hiding principle. We present what are the specifications needed to verify the code in Figure 4 using separation logic.

The predicate for the Producer class is $Prod(this, ss, es, sl, el)$, where :

$Prod(p, ss, es, sl, el) \equiv p.startSmallJobs \to ss \ \star \ p.endSmallJobs \to es \ \star$
$p.startLargeJobs \to sl \ \star \ p.endLargeJobs \to el.$

The precondition for the `produce()` method is:

$Prod(p, ss, es, sl, el) \ \star \ Listseg(ss, null, 0, 10) \ \star \ Listseg(sl, null, 11, 100).$

The predicate for the Consumer class is

$Cons(c, s) \equiv c \to s.$

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

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

The predicate for the Control class is :

$Ctrl(ct, p1, p2, c1, c2) \equiv ct.prod1 \to p1 \ \star \ ct.prod2 \to p2 \ \star$
$ct.cons1 \to c1 \ \star \ ct.cons2 \to c2.$

The precondition for `makeActive()` is:

$Ctrl(this, p1, p2, c1, c2) \ \star \ Prod(p1, ss, es, sl, el) \ \star \ Prod(p2, ss, es, sl, el) \ \star$
$Cons(c1, sl) \star Cons(c2, ss) \star Listseg(ss, null, 0, 10) \star Listseg(sl, null, 11, 100).$

The lack of modularity will manifest itself when we add the two queues as in the right image of Figure 4.

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

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

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

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

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

$Ctrl(this, p1, p2, c1, c2) \star Prod(p1, ss1, es1, sl1, el1) \star Prod(p2, ss2, es2, sl2, el2) \star$
$Cons(c1, sl1, sl2) \ \star \ Cons(c2, ss1, ss2) \ \star \ Listseg(ss1, null, 0, 10) \ \star$
$Listseg(ss2, null, 0, 10) \star Listseg(sl1, null, 11, 100) \star Listseg(sl2, null, 11, 100)$

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

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

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

  public void produce()
   { Random generator = new Random();
   int r = generator.nextInt(101);
   Link l = new Link(r, null);
   if (r <= 10)
   {   if (startSmallJobs == null)
         { startSmallJobs = l;
          endSmallJobs = l;}
    else
       {endSmallJobs.next = l;
         endSmallJobs= l;}
   }
   else
   {   if (startLargeJobs == null)
         { startLargeJobs = l;
          endLargeJobs = l;}
    else
       {endLargeJobs.next = l;
        endLargeJobs = l;}
   }
  }
}
```

Fig. 5. Producer class

```
public class Consumer {

      Link startJobs;

  public Consumer(Link s) {
        startJobs = s;

  public void consume()
    { if (startJobs != null)
       {System.out.println(startJobs.val);
       startJobs = startJobs.next;}
  }
```

Fig. 6. Consumer class

```
public class Control {
      Producer prod1, prod2;
      Consumer cons1, cons2;

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

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

Fig. 7. Control class

1.4 Proposed Approach

This thesis proposes a method for modular verification of object-oriented code in the presence of aliasing. Through the use of *object propositions*, I am able to hide the shared data that two objects have in common. The implementations of the two objects have a shared fractional permission [13] to access the common data, but this need not be exposed in their external interface. My solution is therefore more modular in some cases than the state of the art with respect to hiding shared data, and furthermore shares strong technical similarities with systems for which there is good automated tool support [10]. This thesis will present a full implementation of the object propositions methodology.

2 This Thesis

The aim of this thesis is to enrich the world with a practical verification tool for object-oriented programs in single-threaded settings. The contributions will be three-fold: the theory presenting the proof rules and the proof of soundness, the implementation of the methodology in the form of an Eclipse plugin named Oprop, and a user study showing the usefulness of the tool.

2.1 Thesis Statement

Object propositions, which statically characterize both the aliasing behavior of program references in object-oriented programs and the abstract predicates that hold about the objects, can be successfully used in single-threaded object-oriented programs to write specifications and to prove that those specifications are obeyed by the code.

2.2 Hypotheses

We can break the thesis statement down into more concrete and measurable hypotheses.

Hypothesis:Formalization We can develop and formalize a system that will guarantee that a single-threaded program including formal specifications obeys those specifications. If the specifications are not respected, the proof system can be used to signal which part of the specifications are being broken.

Validation This hypothesis is validated by the development and formalization of my type system and dynamic semantics based on object propositions. I have already proven that the type system is sound with respect to its semantics. The soundness of the proof rules means that given a heap that satisfies the precondition formula, a program that typechecks and verifies according to my proof rules will execute, and if it terminates, will result in a heap that satisfies the postcondition formula.

Hypothesis:Practicality My verification system can be used to verify scientifically significant object oriented programs.

Validation In order to validate this hypothesis, I will use the Eclipse plugin that I will implement to verify 10 small and 2-4 more complex Java programs. I will choose programs that include updates to shared data structures that have multi-object invariants. The complex Java programs will represent more complicated implementations, such as the composite pattern that has already been verified by hand with the help of my verification system.

All programs have to be written such that their syntax matches the syntax of programs that can be verified using Oprop. I am going to incorporate as many Java features as possible in the syntax of the object proposition methodology, but I am not going to be able to incorporate all Java features.

The small programs that I am going to verify have to be scientifically significant. This can be achieved in the following way: by finding errors that are common in practice and by writing programs that illustrate those errors. This kind of common errors can be found by surveying Java forums on the Internet and seeing what are the most common problems that developers post. My wish is for Oprop to discover the errors that I am going to inject in the small programs. We could go even further and look for more rare errors that developers post on Internet forums. When these errors are reproduced in programs, my desire is for Oprop to discover them.

An alternative route that is equally interesting is to take sample pieces of code from well-known open source projects and verify them using Oprop. This exercise would show how Oprop deals with real code and it would be an exciting experiment.

Hypothesis:Usability My tool is usable by programmers and the specifications lead to a better understanding of the aliasing patterns in the analyzed programs, with the cost of small additional overhead represented by the burden of formal specifications.

Validation I will evaluate this hypothesis through a user study. I will show that developers successfully use my Eclipse plugin to find bugs in their programs. Additionally, I will show that the specifications written using object propositions facilitate the understanding of programs and of the aliasing patterns in those programs. This will be especially important for developers who are not familiar with the code beforehand.

3 Theoretical contributions of the thesis

The main theoretical contributions of my thesis are the following:

- 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 [13], fractions do not restrict mutation of the shared data; instead, they require that the specified invariant be preserved.
– A proof of soundness in support of the system.
– Validation of the approach by specifying and proving partial correctness of the composite pattern, demonstrating benefits in modularity and abstraction compared to other solutions with the same code structure.

A big contribution of my work is related to modularity. My work is modular to a class and it allows multiple objects that are part of the same data structure to be modified simultaneously.

Below I highlight the main differences in modularity compared to existing approaches:

– like separation logic and permissions, but unlike conventional object invariant and ownership-based work (including [31] and [32]), my 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, I can modify objects without owning them. More broadly, unlike either ownership or separation logic systems, in my system object A can depend on a 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.
– unlike concurrency approaches, in our system shared data need not be guarded by a lock

4 Description of proposed approach

My methodology uses abstract predicates [34] to characterise the state of an object, embeds those predicates in a logical framework, and specifies sharing using fractional permissions [13].

My main technical contribution is the novel abstraction called *object proposition* that combines predicates with aliasing information about objects. Object propositions combine 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 solution is therefore more modular than the state of the art with respect to hiding shared data, and furthermore generalizes systems [10] for which there is good automated tool support.

My checking approach is modular and verifies that implementations follow their design intent. In my 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 (including for fields of primitive type *int*). Following Fähndrich and DeLine [19], 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. At the end of a public method, we *pack* [16] the keys back into an object proposition and check that object proposition against the method post-condition.

4.1 Examples with Object Propositions

Cells in a spreadsheet In Figure 9, I present the Java class from Figure 8 augmented with predicates and object propositions, which are useful for reasoning about the correctness of client code and about whether the implementation of a method respects its specification. Since they contain fractional permissions which represent resources that have to be consumed upon usage, the object propositions are consumed upon usage and their duplication is forbidden. Therefore, I use linear logic [21] 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).

Newly created objects have a fractional permission of 1, and their state can be manipulated to satisfy different predicates defined in the class. A fractional permission of 1 can be split into two fractional permissions which are less than 1, see Figure 18. The programmer can specify 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.

A critical part of my work is allowing clients to depend on a property of a shared object. Other methodologies such as Boogie [7] allow a client to depend only on properties of objects that it owns. My verification technique also allows a client to depend on properties of objects that it doesn't (exclusively) own.

To gain read or write access to the fields of an object, we have to *unpack* it [16]. 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 in Section 6 require us to 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. We avoid inconsistencies by allowing multiple object propositions to be unpacked at the same time only if the objects are not aliased, or if the unpacked propositions cover disjoint fields of a single object.

Packing/unpacking [16] is a very important mechanism in my system. The benefits of this mechanism are the following:

```
class Dependency {
      Cell ce;
      int input;
 }
class Cell {
      int in1, in2, out;
      Dependency dep1, dep2;

  void setInputDep(int newInput) {
    if (dep1!=null) {
          if (dep1.input == 1) dep1.ce.setInput1(newInput);
          else dep1.ce.setInput2(newInput);
    }
    if (dep2!=null) {
          if (dep2.input == 1) dep2.ce.setInput1(newInput);
          else dep2.ce.setInput2(newInput);
    }
    }

  void setInput1(int x) {
    this.in1 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
    }

  void setInput2(int x) {
    this.in2 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
    }

 }
```

Fig. 8. Cell class

```
class Dependency {
      Cell ce;
      int input;

  $predicate\ OKdep(int\ \ o)\ \ \equiv this.ce \rightarrow c \otimes this.input \rightarrow i \otimes$
     $\exists k1, y1, y2\ \ .\ \ c@k1\ \ OK(y1, y2)\ \ \otimes\ \ c@1\ \ Input(y1, y2, i, o)$
 }
class Cell {
      int in1, in2, out;
      Dependency dep1, dep2;

  $predicate\ \ Input(int\ \ x1, int\ \ x2, int\ \ i, int\ \ x)\ \ \equiv (i = 1 \otimes x1 = x) \oplus (i = 2 \otimes x2 = x)$

  $predicate\ \ OK()\ \ \equiv \exists x1, x2, o, d1, d2. this.in1 \rightarrow x1 \otimes this.in2 \rightarrow x2\ \ \otimes$
      $this.out \rightarrow o \otimes this.dep1 \rightarrow d1 \otimes this.dep2 \rightarrow d2 \otimes x1 + x2 = o\ \ \otimes$
     $d1@1\ \ OKdep(o) \otimes d2@1\ \ OKdep(o)$

  void setInputDep(int i, int newInput) {
    if (dep1!=null) {
          if (dep1.input == 1) dep1.ce.setInput1(newInput);
          else dep1.ce.setInput2(newInput);
    }
    if (dep2!=null) {
          if (dep2.input == 1) dep2.ce.setInput1(newInput);
          else dep2.ce.setInput2(newInput);
    }
    }

  void setInput1(int x)
  $\exists k.(this@k\ \ OK() \multimap this@k\ \ OK())$
  { this.in1 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
  }

  void setInput2(int x)
  $\exists k.(this@k\ \ OK() \multimap this@k\ \ OK())$
  { this.in2 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
  }

 }
```

Fig. 9. Cell class and OK predicate

- it achieves information hiding (e.g. like abstract predicates)
- it describes the valid states of the system (similar to visible states in invariant-based approaches)
- it is a way to store resources in the heap. When a field key is put (packed) into a predicate, it disappears and cannot be accessed again until it is unpacked
- it allows us to characterize the correctness of the system in a simple way when everything is packed

Another central idea of my system is *sharing* using fractions less than 1. The insights about sharing are the following:

- with a fractional permission of 1, no sharing is permitted. There is only one of each abstract predicate asserted for each object at run time, and the asserted abstract predicates have disjoint fields.
- fractional permissions less than 1 enable sharing of particular abstract predicates, but only one instance of a particular abstract predicate P on a particular object o can be unpacked at once. This ensures that field permissions cannot be duplicated via shared permissions.

An important aspect of my system is the ability to allow predicates to depend on each other. Intuitively, this allows "chopping up" an invariant into its modular constituent parts.

Like other previous systems, my system uses abstraction, which allows clients to treat method pre/post-conditions opaquely.

The predicate *OK()* in Figure 9 ensures that all the cells in the spreadsheet are in a consistent state, where the sum of their inputs is equal to their output. Since we only use a fractional permissions $k1, k2 < 1$ for the dependency cells, it is possible for multiple predicates $OK()$ to talk about the same cell without exposing the sharing. More specifically, using object propositions we only need to know $a1@k\ OK()$ before calling $a1.setInput1(10)$. Before calling $a2.setInput1(20)$ we only need to know $a2@k\ OK()$. Since inside the recursive predicate $OK()$ there are fractional permissions less than 1 that refer to the dependency cells, we are allowed to share the cell $a3$ (which can depend on multiple cells). Thus, using object propositions we are not explicitly revealing the shared cells in the structure of the spreadsheet.

Simulator for Queues of Jobs The code in Figures 11, 12 and 13 represents the code from Figures 5, 6 and 7, augmented with predicates and object propositions. The predicates and the specifications of each class explain how the objects and methods should be used and what is their expected behavior. For example, the Producer object has access to the two queues, it expects the queues to be shared with other objects, but also that the elements of one queue will stay in the range [0,10], while the elements of the second queue will stay in the range [11,100]. Predicate *Range* is defined in Figure 10.

When changing the code to reflect the modifications in the right image of Figure 4, the internal representation of the predicates changes, but their external

```
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]

 }
```

Fig. 10. Link class and Range predicate

semantics stays the same; the producers produce jobs and they direct them to the appropriate queue, each consumer accesses only one kind of queue (either the queue of small jobs or the queue of big jobs), and the controller is still the manager of the system. The predicate `BothInRange()` of the Producer class is exactly the same. The predicate `ConsumeInRange(x,y)` of the Consumer class changes to

`ConsumeInRange(x,y)` $\equiv$ `startJobs1`$\rightarrow o_1 \otimes$ `startJobs2`$\rightarrow o_2$
$\otimes \exists k_1.o_1@k_1$ `Range(x,y)` $\otimes \exists k_2.o_2@k_2$ `Range(x,y)`.

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

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

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

  predicate BothInRange() $\equiv$
    $\exists o_1, o_2$. startSmallJobs$\rightarrow o_1$
         $\otimes$ startLargeJobs$\rightarrow o_2$
         $\otimes$ $\exists k_1.o_1@k_1$ Range(0,10)
         $\otimes$ $\exists k_2.o_2@k_2$ Range(11,100)

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

  public void produce()
   $\exists$ k.this@k BothInRange() $\multimap$
     $\exists$ k.this@k BothInRange() {
   Random generator = new Random();
   int r = generator.nextInt(101);
   Link l = new Link(r, null);
   if (r <= 10)
   {   if (startSmallJobs == null)
         { startSmallJobs = l;
          endSmallJobs = l;}
    else
       {endSmallJobs.next = l;
         endSmallJobs= l;}
   }
   else
   {   if (startLargeJobs == null)
         { startLargeJobs = l;
          endLargeJobs = l;}
    else
       {endLargeJobs.next = l;
        endLargeJobs = l;}
   }
  }
}
```

Fig. 11. Producer class

```
public class Consumer {
      Link startJobs;

  predicate ConsumeInRange(int x, int y) $\equiv$
    startJobs$\rightarrow o$ $\otimes$ $\exists$ k.o@k Range(x,y)

  public Consumer(Link s) {
        startJobs = s;

  public void consume()
    $\forall$ x:int, y:int.
      $\exists$ k.this@k ConsumeInRange(x,y)
        $\multimap$ $\exists$ k.this@k ConsumeInRange(x,y)
    { if (startJobs != null)
       {System.out.println(startJobs.val);
       startJobs = startJobs.next;}
  }
```

Fig. 12. Consumer class

```
public class Control {
      Producer prod1, prod2;
      Consumer cons1, cons2;

  predicate WorkingSystem() $\equiv$
    prod1$\rightarrow o_1\otimes$ prod2$\rightarrow o_2$
         $\otimes$ cons1$\rightarrow o_3\otimes$ cons2 $\rightarrow o_4$
         $\otimes$ $\exists k_1.o_1@k_1$ BothInRange()
         $\otimes$ $\exists k_2.o_2@k_2$ in BothInRange()
         $\otimes$ $\exists k_3.o_3@k_3$ in ConsumeInRange(0,10)
         $\otimes$ $\exists k_4.o_4@k_4$ in ConsumeInRange(11,100)

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

  public void makeActive( int i)
   $\exists k.this@k$ WorkingSystem() $\multimap$
       $\exists k.this@k$ in WorkingSystem() {
    Random generator = new Random();
    int r = generator.nextInt(4);
    if (r == 0) {prod1.produce();}
      else if (r == 1) {prod2.produce();}
        else if (r == 2) {cons1.consume();}
          else {cons2.consume();}
    if (i > 0) { makeActive(i-1);}
        }
    }
```

Fig. 13. Control class

4.2 Example: Simple Composite

The code for a very simple specification of the composite pattern is given below.

```
class Composite {
      Composite left, right, parent;
      int count;

  void setLeft(Composite l) {
    l.parent=this;
    this.left=l;
    this.count=this.count+l.count+1;
  }

  void setRight(Composite r) {
    r.parent=this;
    this.right=r;
    this.count=this.count+r.count+1;
  }

 }
```

Fig. 14. Composite class

The predicates for this class are given in Figure 15.

With the help of those predicates, the specification of the method *setLeft* is written as follows:

$\exists c_1, c_2.this@1\ count(c_1) \otimes l@1\ count(c_2) \multimap this@1\ count(c_1 + c_2 + 1)$.

Below there is an example that can be verified using separation logic, but can also be verified using object propositions.

```
...
   {}
Composite a = new Composite();
   {a@1 count(0)}
Composite b = new Composite();
   {b@1 count(0)}
a.setLeft(b);
   {a@1 count(1) ⊗ b@1 count(0)}
a.setRight(c);
   {a@1 count(2) ⊗ b@1 count(0) ⊗ c@1 count(0)}
...
```

As can be seen in the example above, all the fractional permissions are equal to 1, meaning that there is no sharing of data in this example. Because the data

$$\begin{aligned}\textsf{predicate } \mathit{count}\ (\textsf{int c}) \equiv{} & \exists \textsf{ol, or, lc, rc. this.count} \rightarrow \textsf{c} \otimes \\ & \textsf{c} = \textsf{lc} + \textsf{rc} + 1 \ \otimes \textsf{this}@1\ \mathit{left}(\textsf{ol, lc}) \\ & \otimes\, \mathit{this}@1\ \mathit{right}(\textsf{or, rc})\end{aligned}$$

$$\begin{aligned}\textsf{predicate } \mathit{left}\ (\textsf{Composite ol}, \textsf{int lc}) \equiv{} & \textsf{this.left} \rightarrow \textsf{ol} \otimes \\ & \big((\textsf{ol} \neq \textsf{null} \ \multimap \textsf{ol}@\frac{1}{2}\ \mathit{count}(\textsf{lc})) \\ & \oplus (\textsf{ol} = \textsf{null} \ \multimap\ \textsf{lc} = 0)\big)\end{aligned}$$

$$\begin{aligned}\textsf{predicate } \mathit{right}\ (\textsf{Composite or}, \textsf{int rc}) \equiv{} & \textsf{this.right} \rightarrow \textsf{or} \otimes \\ & \big((\textsf{or} \neq \textsf{null} \ \multimap \textsf{or}@\frac{1}{2}\ \mathit{count}(\textsf{rc})) \\ & \oplus (\textsf{or} = \textsf{null} \ \multimap\ \textsf{rc} = 0)\big)\end{aligned}$$

Fig. 15. Predicates for Simple Composite

structure does not have sharing of data and there is no danger of exposing any shared data, it is very suitable to verification using separation logic. But it can also be verified using object propositions. The notion of a fractional permission of 1 is incorporated in object propositions, which basically means that data is not shared and can be changed only from one reference point.

5 Formal System

5.1 Grammar

The programming language that I are using is inspired by Featherweight Java [22], extended to include object propositions. I retained only Java concepts relevant to the core technical contribution of this paper, omiting features such as inheritance, casting or dynamic dispatch that are important but are handled by orthogonal techniques.

Below I show the syntax of my simple class-based object-oriented language. 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. References o and indirect references l do not appear in source programs but are used in the dynamic semantics, defined later. In the grammar, r represents a reference to an object and i represents a reference to an integer.

$$\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\ x}}\texttt{)} \equiv \mathsf{R} \\
\mathsf{MthDecl} & ::= & \mathsf{T}\ m\texttt{(}\overline{\mathsf{T\ x}}\texttt{)}\ \mathsf{MthSpec}\ \texttt{\{}\ \bar{\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 \bar{z}.\mathsf{R}\ \mid\ \forall \bar{z}.\mathsf{R}\ \mid\ r.f \rightarrow \mathsf{x}\ \mid\ \mathsf{t\ binop\ t} \\
\mathsf{P} & ::= & r@\mathsf{k}\ Q\texttt{(}\bar{\mathsf{t}}\texttt{)}\ \mid\ \texttt{unpacked(}r@\mathsf{k}\ Q\texttt{(}\bar{\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\text{)} \\
\mathsf{e} & ::= & \mathsf{t}\ \mid\ r\texttt{.}f\ \mid\ r\texttt{.}f\ \texttt{=}\ \mathsf{t}\ \mid\ r\texttt{.}m\texttt{(}\bar{\mathsf{t}}\texttt{)}\ \mid\ \texttt{new}C\texttt{(}\bar{\mathsf{t}}\texttt{)}\ \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\texttt{(}\bar{\mathsf{t}}\texttt{)in e}\ \mid\ \texttt{unpack}\ r@\mathsf{k}\ Q\texttt{(}\bar{\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\ !=\ \mid\ \leq\ \mid\ <\ \mid\ \geq\ \mid\ > \\
\mathsf{T} & ::= & C\ \mid\ \texttt{int}\ \mid\ \texttt{boolean}
\end{array}$$

In my system, I will assume that all the formulas R are in disjunctive normal form. A formula R of my system is in disjunctive normal form if and only if it is an additive disjunction of one or more multiplicative conjunctions of one or more of the predicates P , t_1 `binop` t_2 , $r.f \rightarrow x$, $\exists z.\mathsf{P}$, $\forall z.\mathsf{P}$.

5.2 Permission Splitting

In order to allow objects to be aliased, we must split a fraction of 1 into multiple fractions less than 1 [13]. The fraction splitting rule is defined in Figure 18. An invariant of the rules is that a fraction of 1 is never duplicated. We also allow the inverse of splitting permissions: joining, where we define the rules in Figure 19.

6 Proof Rules

This section describes the proof rules that can be used to verify correctness properties of code. The judgement 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 judgement 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 pre-condition in disjunctive normal form. Π should be just as general as R.

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

The static proof rules also contain the following judgements: $\Gamma \vdash r : C$, $\Gamma; \Pi \vdash R$ and $\Gamma; \Pi \vdash r.T; R$. The judgement $\Gamma \vdash r : C$ means that in valid type context Γ, the reference r has type C. The judgement $\Gamma; \Pi \vdash R$ means that from valid type context Γ and linear context Π we can deduce that object proposition R holds. The judgement $\Gamma; \Pi \vdash r.T; R$ 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. The $\otimes$ linear logic operator is symmetric. Thus in the rules for adding fractions, we can have a rule symmetric to (ADD2) that adds the fraction of a packed object propositions to the fraction of an unpacked object proposition.

Additionally in 16 I present a deductive rule for the principle that if we have two permissions with the same name to the same object, then their arguments must be equal. In 17 I present a rule that is needed for reasoning by contradiction. Both these rules are used in the proof of the Composite pattern that I present in section 10.

$$\frac{r@k\ Q(\overline{t_1}) \quad r@k\ Q(\overline{t_2})}{\overline{t_1} = \overline{t_2}}\ (\textsc{SameArgs})$$

Fig. 16. Rule for predicates with same arguments

$$\frac{Q(\overline{t}) \quad (!Q(\overline{t}))}{false}\ (\textsc{Contr1})$$

$$\frac{r@k_1\ Q(\overline{t_1}) \qquad r@k_2\ Q(\overline{t_1}) \qquad k_1 + k_2 > 1}{false}\ (\textsc{Contr2})$$

Fig. 17. Rules for reasoning about contradiction

Before presenting the detailed rules, I provide the intuition for why my system is sound (the formal soundness theorem is given below, and proved in the supplemental material). The first invariant enforced by my 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 fractional permissions less than 1, follows an inductive argument. 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

$$\frac{k \in (0,1]}{r@k\ Q(\bar{t}) \vdash\ r@\frac{k}{2}\ Q(\bar{t}) \otimes r@\frac{k}{2}\ Q(\bar{t})}\ (\textsc{Split})$$

Fig. 18. Rule for splitting fractions

$$\frac{\epsilon \in (0,1) \qquad k \in (0,1] \qquad \epsilon < k}{r@\epsilon\ Q(\overline{t_1}) \otimes r@(k-\epsilon)\ Q(\overline{t_2}) \vdash\ r@k\ Q(\overline{t_1})}\ (\textsc{Add1})$$

$$\frac{\epsilon \in (0,1) \qquad k \in (0,1] \qquad \epsilon < k}{\begin{array}{c}\mathit{unpacked}(r@\epsilon\ Q(\overline{t_1})) \otimes r@(k-\epsilon)\ Q(\overline{t_2}) \vdash \\ \mathit{unpacked}(r@k\ Q(\overline{t_1}))\end{array}}\ (\textsc{Add2})$$

Fig. 19. Rules for adding fractions

being shared) always holds whenever that object is 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 an object, we must first unpack it; by induction, we can assume its invariant holds as long as the object is packed. But we know the object is packed immediately before the unpack operation, because the rules of my 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 invariant 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. 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 proof rules while inlining the rules in the text.

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

$$\frac{\Gamma \vdash t : T \quad \Gamma; \Pi \vdash [t/x]R}{\Gamma; \Pi \vdash t : \exists x.T; R} \textsc{Term}$$

$$\frac{\begin{array}{c}\Gamma \vdash r : C \quad r.f_i : T \ \mathit{is\ a\ field\ of}\ C \\ \Pi \vdash r.f_i \to r_i \qquad \Gamma; \Pi \vdash [r_i/x]R\end{array}}{\Gamma; \Pi \vdash r.f_i : \exists x.T; R} \textsc{Field}$$

New checks object construction. We get a key for each field and the remaining linear context from which we consumed the keys.

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

If introduces disjunctive types in the system and checks *if*-expressions. A corresponding $\oplus$ rule eliminates disjunctions in the pre-condition by verifying that an expression checks under either disjunct.

$$\frac{\begin{array}{c}\Gamma; (\Pi \otimes t = \texttt{true}) \vdash e_1 : \exists x.T; R_1 \\ \Gamma \vdash t : \texttt{bool} \quad \Gamma; (\Pi \otimes t = \texttt{false}) \vdash e_2 : \exists x.T; R_2\end{array}}{\Gamma; \Pi \vdash \ \texttt{if(t)\{}e_1\texttt{\}else\{}e_2\texttt{\}} : \exists x.T; R_1 \oplus R_2} \textsc{If}$$

Let checks a *let* binding, extracting existentially bound variables and putting them into the context (a limitation of my current system is that universal quantification is supported only in method specifications).

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash e_1 : \exists x.T_1; R_1 \otimes \Pi_2 \\ (\Gamma, x : T_1); (R_1 \otimes \Pi_2) \vdash e_2 : \exists w.T_2; R_2\end{array}}{\Gamma; \Pi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \exists w.T_2; R_2} \textsc{Let}$$

$$\frac{\Gamma; \Pi_1 \vdash e : \exists x.T; R_1 \qquad \Gamma; \Pi_2 \vdash e : \exists x.T; R_2}{\Gamma; (\Pi_1 \bigoplus \Pi_2) \vdash e : \exists x.T; R_1 \oplus R_2} \oplus$$

The Call rule simply states what is the object proposition that holds about the result of the method being called. This rule first identifies the specification of the method (using the helper judgement Mtype) and then goes on to state the object proposition holding for the result. The $\vdash$ 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 are unique. The normal situation is that all shared predicates are packed, and any method can be called in this situation. In the intended mode of use, we only make calls with a shared unpacked predicate when traversing a data structure hand-over-hand as in the Composite pattern, and we claim that modularity problems are minimized in this situation. This does represent a limitation in our system, however, it is one that goes hand in hand with the advantage of supporting shared predicates.

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

$$\frac{\begin{array}{c} class\ C\{...\overline{M}...\} \in \overline{CL} \\ T_r\ m(\overline{Tx})R_1 \multimap R\ \{\overline{e_1};\ return\ e_2\} \in \overline{M}\end{array}}{mtype(m, C) = \forall \overline{x : T}.\exists result.T_r; R_1 \multimap R} \ \textsc{Mtype}$$

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.

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

The rules for packing and unpacking are PACK1, PACK2, UNPACK1 and UNPACK2. When we pack an object to an 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 that the one that was true for them before unpacking.

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

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

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. In the rules below we assume that there is a class C that is fixed.

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

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

I have also developed rules for the dynamic semantics, that are used in proving the soundness of my system. The dynamic semantics rules can be found in the technical report.

7 Implementation

7.1 Object Proposition Inference Algorithm

The proof rules presented in the section 6 are deterministic because at each point in the program there is a unique rule that can be applied. The proof rules have not been written in this deterministic form from the beginning. For example, the rule PACK2 was first written as below:

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; [\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \qquad 0 < k < 1 \\ \Gamma; (\Pi' \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \otimes \Pi' \vdash \texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack2}$$

The above rule is not deterministic because there is a context split that has to be guessed. The tool Oprop would have to guess which part of the linear context in the conclusion is Π and which part is Π'. We have rewritten the PACK2 rule to make it deterministic and to eliminate the need to split contexts. The current form of the rule is:

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

In order to manage context splits, we have used resource management techniques for linear proof search [14].These techniques add an additional output, the "leftover" resources, to judgements. The idea of resource management is that we can make context splits deterministic by sending all resources for proving the first premise, then use the leftover resources to prove the second premise, then use the leftover resources from the second premise to prove the third premise, and so on.

Another source of non-determinism is that object propositions can be split an arbitrary number of times and merged back together. The formal system "guesses" exactly how a given permission inside of an object proposition has to be split up and merged in order to satisfy different object propositions (for example in order to satisfy a pre-condition). For example, a fractional permission of 1 can be split into two fractions $\frac{1}{4}$ and $\frac{3}{4}$, or into two fractions of $\frac{1}{2}$. In this case, we would have the following constraints C for the fractions k_1 and k_2: $k_1 + k_2 = 1$, $k_1, k_2 \in (0, 1)$. We track constraints C together with the current linear context Π. The constraints accumulated during the verification procedure allow us to know everything about the permissions needed inside a piece of code. In order to prove that a program is correct, we will have to prove that the constraints are satisfiable. As discussed in [9], Fourier-Motzkin elimination can be used to check the satisfiability of the fractional constraints.

One final source of non-determinism comes from situations where multiple proof rules for linear logic are applicable. For example, for proving the choice $P_1 \oplus P_2$ there are two linear logic rules that can be applied:

$$\frac{\Gamma; \Pi \vdash P_1}{\Gamma; \Pi \vdash P_1 \oplus P_2} \oplus_L \quad \frac{\Gamma; \Pi \vdash P_2}{\Gamma; \Pi \vdash P_1 \oplus P_2} \oplus_R$$

There are two options when multiple proof rules can be applied: we either can use backtracking or we can use forward reasoning. Backtracking would mean that we arbitrarily choose one of the applicable rules and if it turns out that the chosen rule does not work, we roll back and try the other one.

In our setting, backtracking is not the best solution. We are trying to verify an entire program. For doing this, we have to prove linear logic predicates for every method call and object construction. Whenever such a proof does not succeed, we would have to backtrack to the last call or construction site in the program where we made a choice. Similarly to [9], our tool is going to implement a dataflow analysis. Backtracking to the last call or construction site as part of a dataflow analysis is difficult because we would have to roll back the entire state of the flow analysis. Due to these reasons, we choose to carry all possible choices forward.

8 The Oprop Eclipse Plugin

The tool that I am going to implement will be called Oprop and will be a plug-in to the Eclipse IDE. The tool represents the implementation of the object proposition system as a static dataflow analysis for Java. In the remainder of this section I discuss practical details of the implementation and other existing Java packages and plug-ins that are going to help with the implementation. The Oprop tool will be able to verify the correctness of single-threaded programs. The extension of Oprop into a tool that can verify multi-threaded programs is left as future work.

In order to specify method pre- and post-conditions using object propositions, developers will need to use annotations that can express linear logic concepts and fractional permissions. A Java package needs to be implemented to express the syntax of object propositions. The classes in that package will be used to write object propositions such as those in Figure 9. Annotations will be the main way in which developers interact with Oprop.

8.1 Eclipse

Eclipse [4] is a software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. Since Eclipse provides the infrastructure to develop plug-ins that improve the standard Eclipse capabilities, we are going to use Eclipse both to write the Oprop plugin and as the target of our plug-in. We are going to use the following Eclipse component: the Java AST.

Eclipse offers a parser, typechecker and abstract syntax tree (AST) for Java source files. This might help Oprop with the parsing, typechecking and representing of Java code, which could simplify my work. Oprop will only handle a Java subset and I might not be able to use the Eclipse parser, but the Eclipse parser could give me some insight into how to construct my own.

8.2 Implementation of Oprop

Abstract predicates can be used to express properties of fields using integers, but reasoning about integers is difficult. We are going to employ a theorem prover to prove properties about integers. Most probably, we are going to use the Z3 theorem prover [3].

Kevin Bierhoff has implemented a tool called JavaSyp [1] that uses the SMT solver Z3 to formally verify Java code. We initially wanted to modify JavaSyp in order to implement Oprop, but that proved to be a difficult task. There are many details that are different between JavaSyp and what Oprop needs to do:

- JavaSyp uses borrowing and capture/release because the tool does not implement fractional permissions. Oprop does not use borrowing, but instead it uses fractional permissions. Fractions give more precision than the borrowing mechanism and we are going to implement fractional permissions.
- Oprop will implement the pack/unpack mechanism, while JavaSyp does not implement this mechanism. JavaSyp implements instead “exposure blocks" that show how fields should be accessed. These features are closely related: when the fields of an object are unpacked (when the object proposition that encapsulated them is unpacked), we can think of them as being “exposed". In Bierhoff’s system, there are “unique" and “immutable" exposure blocks. Fields can be assigned inside unique exposure blocks, with field reads yielding the field’s original permission. Inside an immutable block, reading fields results in a weakened field permission. The technical difference is that in my system I do not have immutable permissions, but instead one can always write to the fields of an object (in some cases, one has to make sure that the invariant is preserved). I acknowledge that this is just an incidental difference and the ideas of pack/unpack vs. exposure blocks are very similar.
- JavaSyp has class invariants, but Oprop will not have them and instead it will have invariants for objects that are shared.

Our plan is to create a tool similar to JavaSyp that first translates the code and specifications into an intermediary language (such as those used by Dafny or Chalice, from the RiSE group at Microsoft Research [7]) and then use that intermediary tool together with Z3 to obtain the final result.

9 Validation and Evaluation

We will evaluate the usefulness of the Oprop tool and of the object proposition methodology by designing a user study where we observe how the Oprop plug-in improves the efficiency of programmers. We will also try to solve the open problem **Finalizers**, that will be described below, using our methodology. We have already made some contributions to the specification and verification of the Composite pattern, and we will finalize the verification of the Composite pattern.

9.1 User Study

The primary purpose of the user study is to see if programmers can use the Oprop plugin to find errors in their code. A secondary purpose is to see if the specifications help programmers that are not familiar with the code to better understand the meaning of the code and the aliasing patterns. In this sense, one could say that the specifications act like documentation.

The first step would be to teach the programmers how to use the object proposition methodology. This would be done in a two hour seminar. Next, we would present the programmers with multiple programs that contain errors. We would observe how the programmers use Oprop and take notes about their experience using Oprop.

Additionally, we would show to a first set of developers some new programs (that the developers have not seen before) that do not contain any annotations and see how well they understand the meaning of the programs. Next, we would show to a second set of developers the same programs to which we add the object propositions annotations. We would see how well the second set of programmers understand the programs and if the annotations facilitate the understanding of code. Additionally, we could also use one set of programmers and instruct them to think aloud in order to see if they are using the object propositions annotations. The challenge is to find sets of programmers that have similar knowledge and skills, so that we can compare the findings from the different sets. We believe that we can find such sets of programmers.

9.2 Finalizers

This open problem was presented in [29] and I restate it here.

"Finalizers are special methods that are invoked by the runtime system before an unreachable object is de-allocated. Their purpose is mainly to free system resources. For instance, the finalize method in Figure 20 closes the files used by its receiver object. Since the runtime environment of languages like Java and C# may invoke the garbage collector in any execution state, programs have no control over the execution of finalizers. This leads to two problems for verification.

```
import java.io.*;
public class TempStorage {
   private /*@ nullable @*/ FileReader tempFile;
   private /*@ nullable @*/ FileWriter logFile;
   //@ private invariant tempFile != null && logFile != null;
   public TempStorage() throws IOException {
     tempFile = new FileReader("/tmp/dummy");
     logFile = new FileWriter("/tmp/log");
   }
   protected void finalize() throws Throwable {
     super.finalize();
     logFile.write("Bye bye");
     logFile.close();
     tempFile.close();
   }
}
```

Fig. 20. The *finalize* method closes the files used by the receiver object. Although non-null is the default in JML, we include, for emphasis, a declaration that makes this invariant explicit.

"First, a finalizer might be invoked in a state in which certain object invariants do not hold. In our example, the constructor of TempStorage throws an exception if opening the files fails. In this case, the object is never fully initialized and thus its invariant does not hold. However, the finalizer of TempStorage relies on the object invariant and, therefore, will abort with a null-pointer exception when a partly-initialized object is destroyed. A verification technique can prevent an application program from calling a method on a partly-initialized object, for instance, by making explicit which object invariant may be assumed to hold. However, finalizers are called by the runtime system and, therefore, cannot be controlled.

"Second, like any other method, a finalizer potentially modifies the heap. Since finalizers might be called in any execution state, a verification technique has to deal with spontaneous heap changes, which is even worse than the heap changes caused by static initializers.

"Dealing with these problems is an open challenge: to develop a verification technique for finalizers. A solution to this challenge is necessary to guarantee that verification is not unsound for programs containing finalizers."

We think that we can provide a solution to the Finalizers open problem by using the object proposition methodology.

First, we could use object propositions to make sure that the required invariant holds for the object manipulated by the finalizer. A problem could be that Oprop will statically verify the correctness of programs, while any property that a finalizer requires has to be checked at runtime. We can go around this inconvenience by asserting the finalizer precondition after every statement of the program. This will statically simulate the fact that the finalizer can be called at runtime at any point in the program that we want to verify.

A second possible solution stems from the fact that a finalizer is invoked by the runtime system before an unreachable object is de-allocated. Unlike destructors, finalizers are usually not deterministic [2]. A destructor is run when the program explicitly frees an object. In contrast, a finalizer is executed when the internal garbage collection system frees the object. Depending on the garbage collection method used, this may happen at an arbitrary moment after the object is freed, possibly never. Since the garbage collector frees an object when that object is not reachable anymore, we could do the following: for every object, we determine through a static analysis when that object may not be reachable anymore. This analysis might be imprecise in the following two ways: either an object is not reachable but the analysis reports that it is reachable, or the object is reachable but the analysis states that it is not reachable. The latter case will only increase the number of points in the program where the finalizer might potentially be called. The first case however will make the approach unsound because it will not check that the finalizer holds in all points where it should. Thus the requirement is that the static analysis has to be sound and it finds out when an object is not reachable. Only at that point the garbage collector will free the object and the finalizer will be called. Hence only at that point the pre-condition of the finalizer has to hold for the object that is being freed.

The finalizer will also have its pre-condition written using object propositions and we could verify that the pre-condition is satisfied every time the finalizer is called. The object propositions that appear in the pre-condition depend very much on the body of the finalizer method. For example, the pre-condition of the finalizer in class TempStorage from Figure 20 could ensure that *this* object has both fields $tempFile$ and $logFile$ properly initialized.

According to [29] "Concerning the second problem, it seems necessary to allow a finalizer to modify only those objects and system resources that are exclusively used by the object that is being destroyed. In particular, finalizers must not modify global state such as static fields. Techniques such as ownership type systems may be useful for reasoning about the sharing of objects. However, it is unclear how to guarantee that certain system resources are not shared, for instance, how to prevent two objects from creating handlers for the same file."

We can use fractional permissions to make sure that the finalizer only modifies objects that are exclusively used by the object that is being destroyed. When a finalizer is called on an object, it means that the main program does not have a permission to that object anymore (or it has a permission of the kind $\exists k.obj@k\ pred()$, but this permission is not usable). Thus the finalizer should only be allowed to modify objects to which the main program has no more permissions.

In the special case when there is a circular data structure such as a circular linked list, there will be permissions between the linked cells of the list, but there should be no permissions to the cells of the list from the outside. When analyzing the permissions, it could happen that we erroneously state that there are permissions to the cells of the list and do not acknowledge that these permissions come from the other cells. One way to solve this problem is to pay special attention when we have a circular data structure and make sure that the only permissions that still exist to the data structure come from within it and not from the encompasing code. This is the condition that has to hold for any object that the finalizer modifies. Keeping track of the fractional permissions in the program is possible with the object proposition methodology.

The risk of tackling the verification of finalizers is that we might not be able to write pre-conditions accurate enough for the finalizer. Thus the finalizer would potentially be called in a moment when although the pre-condition holds, the pre-condition is not enough to ensure the well-formedness of the objects in the program. We will be able to asses this risk better once we write the specifications and analyze some programs involving finalizers.

10 Composite

The Composite design pattern [20] 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 [29], 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. [11, 24]). We describe a new formalization and proof of the Composite pattern using fractions and object propositions that provides more local reasoning than prior solutions. For example, in Jacobs et al. [24] 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 solve a practical problem that has been a challenge problem so far: the specification and verification of an instance of the composite pattern. 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. Our specification and verification of the composite pattern allows 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.

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 below.

```
class Composite {
  private Composite left, right, parent;
  private int count;

  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)
    else
      newc = newc + left.count;
    if (this.right == null)
    else
      newc = newc + right.count;
    this.count = newc;
   }

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

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

Each Composite caches the size of its subtrees in a count field, so that a parent's count depends on its children's count. The dependency is in fact recursive, as the parent and right/left child pointers must be consistent. Clients can add a new subtree at any time, to any free position (where the current reference is null). 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.

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 [11]. This verification abstracted counts to an even/odd typestate and relied on non-formalized extensions of a formal system, whereas we have formalized the proof system and provide a full proof in the supplemental material. Our verification proves partial correctness of this version of the Composite pattern.

10.1 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 goals of the specification are to allow clients to add a child to any node of the tree that has no left (or right) child. Since the count field of a node depends on the count fields of its children nodes, inserting a child must not violate the transitive parents' invariants.

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.(Note that the invariant needs to hold irrespective of whether the object propositions are packed or unpacked.)

The predicates of the Composite class are presented in Figure 21.

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 $this@\frac{1}{2}\ left(\mathsf{ol},\ \mathsf{lc}) \otimes this@\frac{1}{2}\ right(\mathsf{or},\ \mathsf{rc})$ we connect lc to the left child (through the $left$ predicate) and rc to the right child (through the *right* predicate).

The predicate $left$ expresses that the predicate $count(lc)$ holds for $this.left$, the left child of *this*. The predicate *right* expresses that the predicate $count(rc)$ holds for *this.right*, the right child of *this*. The permission for the $left$ ($right$)

$$\begin{aligned}
&\text{predicate } \mathit{count}\ (\text{int c}) \equiv \exists \text{ol, or, lc, rc. this.count} \rightarrow \text{c} \otimes \\
&\qquad \text{c} = \text{lc} + \text{rc} + 1 \ \otimes \text{this@}\frac{1}{2}\ \mathit{left}(\text{ol, lc}) \\
&\qquad \otimes\, \mathit{this}@\frac{1}{2}\ \mathit{right}(\text{or, rc})
\end{aligned}$$

$$\begin{aligned}
&\text{predicate } \mathit{left}\ (\text{Composite ol, int lc}) \equiv \text{this.left} \rightarrow \text{ol} \otimes \\
&\qquad \big((\text{ol} \neq \text{null} \ \multimap \text{ol@}\frac{1}{2}\ \mathit{count}(\text{lc})) \\
&\qquad \oplus (\text{ol} = \text{null} \ \multimap \ \text{lc} = 0)\big)
\end{aligned}$$

$$\begin{aligned}
&\text{predicate } \mathit{right}\ (\text{Composite or, int rc}) \equiv \text{this.right} \rightarrow \text{or} \otimes \\
&\qquad \big((\text{or} \neq \text{null} \ \multimap \text{or@}\frac{1}{2}\ \mathit{count}(\text{rc})) \\
&\qquad \oplus (\text{or} = \text{null} \ \multimap \ \text{rc} = 0)\big)
\end{aligned}$$

$$\begin{aligned}
&\text{predicate } \mathit{parent}\ () \equiv \exists \text{op, c, k} < \frac{1}{2}.\ \text{this.parent} \rightarrow \text{op} \otimes \\
&\qquad \text{op} \neq \text{this} \ \otimes \ \text{this@}\frac{1}{2}\ \mathit{count}(\text{c}) \otimes \\
&\qquad \Big(\big(\text{op} \neq \text{null} \ \multimap \text{op@k}\ \mathit{parent}() \otimes \\
&\qquad\quad (\text{op@}\frac{1}{2}\ \mathit{left}(\text{this, c}) \\
&\qquad \oplus \text{op@}\frac{1}{2}\ \mathit{right}(\text{this, c}))\big) \oplus \\
&\qquad (\text{op} = \text{null} \ \multimap \ \mathit{this}@\frac{1}{2}\ \mathit{count}(\text{c}))\Big)
\end{aligned}$$

Fig. 21. Predicates for Composite

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 fractional permission to the *count* predicate (and implicitly to its *count* field) of *this*. The *parent* predicate contains only a fraction of $k < \frac{1}{2}$ to the parent of *this* so that any clients can use the remaining fraction to reference the node and add children to the parent. A client can actually use this to update the parent field, but in order to pack the *parent* predicate, the client has to conform to the well-formedness condition mentioned earlier.

If a new node is added to the tree as the left child of *this*, we need to change the *count* field of *this*. The field left of *this* must be null and the permission with a half fraction has to be acquired by unpacking the *count* predicate of *this*. This requires us to unpack the parent's *left* predicate, which requires the parent's *count* predicate, and so on to the root node. We can only pack it back when the tree is in a well-formed state. As the notification algorithm goes up the tree, from the current node to the root, we successively unpack the predicates corresponding to each node and we pack them back when the tree is well-formed. This ensures that if a new node is added, in order to pack the predicates again, the count fields must be updated and consistent!

The proof of partial correctness of the Composite pattern is presented in the supplemental material. The proof is currently done by hand.

The complete specification for each method is given below:

```
class Composite {
  private Composite left, right, parent;
  private int count;

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

  private void updateCountRec ()
∃ k1. (unpacked(this@ k1 parent()) ⊗
∃ opp, lcc, k<half.unpacked(this@half count(lcc))
⊗this.parent → opp ⊗ opp ≠ this ⊗
((opp ≠ null ⊸ opp@k parent() ⊗
(opp@half left(this, lcc) ⊕ opp@half right(this, lcc))) ⊕
(opp = null ⊸ this@half count(lcc)) ) ⊗
∃ ol, lc, or, rc, lcc'. this.count → lcc ⊗ lcc' = lc + rc + 1 ⊗
this@half left(ol, lc) ⊗ this@half right(or, rc)
⊸ this@k1 parent())
    {...}

  private void updateCount ()
∃ c, c1, c2, nc. unpacked(this@1 count(c)) ⊗
this.count → c ⊗ c = c1 + c2 + 1 ⊗
∃ ol, lc, or, rc. this@half left(ol, lc) ⊗
this@half right(or, rc) ⊗
∃ k1. unpacked(this@k1 parent())
⊸ this@1 count(nc) ⊗
nc = lc + rc + 1 ⊗ ∃ k. unpacked(this@k parent())
      {...}

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

The constructor of the class Composite returns half of the permission for the $left$ and $right$ predicate, and half of a permission to the $parent$ predicate.

The method $updateCountRec()$ takes in a fraction of $k1$ to the unpacked $parent$ predicate and a half fraction to the unpacked $count$ predicate of $this$, 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 $this$. Thus, after calling $updateCount()$, the object $this$ satisfies its $count$ predicate.

The method $setLeft(Composite\ l)$ takes in a fraction to the $parent$ predicate of $this$, a fraction to the $parent$ predicate of l and the $left$ predicate of $this$ with a null argument (saying that the left field of $this$ is null and thus a client can attach a new left child here). The post-condition shows that after calling $setLeft$, some of the permission to the $parent$ predicate of this has been consumed, while the fraction to the predicate $parent$ of l stays the same.

11 Related Work

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

Bierhoff and Aldrich [10] developed access permissions, an abstraction that combines typestate and object aliasing information. Developers use access permissions to express the design intent of their protocols in annotations on methods and classes. Our work is a generalization of their work, as we use object propositions to modularly check that implementations follow their design intent. The typestate [16] formulation has certain limits of expressiveness: it is only suited to finite state abstractions. This makes it unsuitable for describing fields that contain integers and can take an infinite number of values and can satisfy various arithmetical properties. Our object propositions have the advantage that they can express predicates over an infinite domain, such as the integers.

Access permissions allow predicate changes even if objects are aliased in unknown ways. States and fractions [13] capture alias types, borrowing, adoption, and focus with a single mechanism. In Boyland's work, a fractional permission means immutability (instead of sharing) to ensure non-interference of permissions. We use fractions to keep object propositions consistent but track, split, and join fractions in the same way as Boyland. Similary, in [12] the fractional permissions are treated as in Boyland's work: when a fraction is 1, then there is write access, but when a fraction is less than 1, one can only read from the shared resource. This is very different from our work because we allow multiple clients to write to a common resource even if the fractional permission is less than 1. The trick is that those clients can only write to the resource if they maintain an invariant on the resource.

Boogie [7] is a modular reusable verifier for Spec# programs. It provides design-time feedback and generates verification conditions to be passed to an automatic theorem prover. While Boogie allows a client to depend on properties of objects that it owns, we allow a client to depend on properties of objects that it doesn't own, too.

Krishnaswami et al. [27] show how to modularly verify programs written using dynamically-generated bidirectional dependency information. They introduce a ramification operator in higher-order separation logic that explains how local changes alter the knowledge of the rest of the heap. Their solution is application specific, as they need to find a version of the frame rule specifically for their library. Our methodology is a general one that can potentially be used for verifying any object-oriented program.

Nanevski et al. [33] 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 [38] introduce Considerate Reasoning, an invariant-based verification technique adopting a relaxed visible-state semantics. Considerate Reasoning allows distinguished invariants to be broken in the initial states of method executions, provided that the methods re-establish the invariant in the final state. The authors demonstrate Considerate Reasoning based on the Composite pattern and provide the encoding of their technique in the Boogie intermediate verification language [7], facilitating the automatic verification of the Composite pattern specification. Despite the fundamental differences in underlying methodology (visble-state invariants vs. abstract predicates) and logic between Considerate Reasoning and our approach, there are interesting analogies in the specification of the Composite pattern. For instance, the method that triggers the bottom-up traversal of the Composite to update a composite's count field in the Considerate Reasoning specification does not expect the composite invariant in the method's initial state. This is similar to our method update-CountRec() which requires the predicates parent and count to be unpacked.

Cohen et al. [15] use locally checked invariants to verify concurrent C programs. In their approach, each object has an invariant, a unique owner and they use handles (read permissions) to accommodate shared objects. The disadvantage is their high annotation overhead and the need to introduce ghost fields. We do not have to change the code in order to verify our specifications.

Our work uses abstract predicates, similar to the work of Parkinson and Bierman [34] and Dinsdale-Young et al. [17]. The abstraction makes it easy to change the internal representation of a predicate without modifying the client's external view of it. The main mechanism is still separation logic, with its shortcomings. Unlike separation logic, we permit sharing of predicates with an invariant-based methodology. This avoids non-local characterizations of the heap structure, as required (for example) in Bart Jacob's Composition pattern solution [24].

There exist a set of verification methodologies for object-oriented programs in a concurrent setting: [17, 23, 30, 26]. These approaches can express externally imposed invariants on shared objects, but only for invariants that are associated with the lock protecting that object. In many cases, it may be inappropriate to associate such an invariant with the lock: for example, in a singlethreaded setting, there is no such lock. Even in multithreaded settings, a high level lock may protect a data structure with internal sharing, in which case specifying that sharing in the lock would break the modularity of the data structure. Thus, these systems do not provide an adequate solution to the modular verification problem we consider.

12 Future Work

There are two lines of work that would lead to the improvement of Oprop.

First, the features of the target language can be augmented. Now, the target language that is being verified using the object proposition technology is a variant of Featherweight Java. In order for Oprop to become a tool used in industry, the target language would have to be as close to Java as possible. During the implementation of Oprop, I will add enough features in order to be able to verify more practical examples of programs. After the completion of this thesis, I want other contributors to be able to add features to the target language. For this to happen, I will construct Oprop so that it is easy to add new features to the target language. I will also write a thorough documentation for Oprop and write comments in the code of the tool.

Second, this work can be extended for multi-threaded programs. Oprop can only verify single-threaded programs, but I hope that it will be a starting point for writing a tool that verifies multi-threaded Java programs.

13 Research Plan

Below is an estimated time-line for the completion of my thesis work. This time-line begins after my thesis proposal talk. The total estimated time is 18 months.

6 months	Implementation of Oprop
6 months	Running the user study and solving the Finalizers problem
6 months	Writing and defense

14 Conclusion

In conclusion, I propose a verification system that is used for single-threaded object-oriented programs. The object proposition methodology uses abstract predicates and fractional permissions. Developers will interact with the Oprop plug-in that implements the object proposition methodology by writing annotations.

I hope that by using Oprop, developers will be more efficient in writing correct programs.

References

1. http://code.google.com/p/syper/.
2. http://en.wikipedia.org/wiki/finalizer.
3. http://research.microsoft.com/en-us/um/redmond/projects/z3/.
4. http://www.eclipse.org/.
5. http://www.java.com/en/about/.
6. Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM.
7. 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.
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. Kevin Bierhoff. Api protocol compliance in object-oriented software. In *PhD thesis. Technical Report CMU-ISR-09-108*, 2009.
10. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007.
11. Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the composite design pattern. In *Proc of SAVCBS 2008*, 2008.
12. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
13. John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003.

14. Iliano Cervesato, Joshuas. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81. Springer-Verlag LNAI, 1996.
15. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.
16. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
17. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
18. Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, pages 213–226, 2008.
19. Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
20. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
21. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.
22. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. pages 132–146, 2001.
23. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.
24. Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Proc of SAVCBS 2008*, 2008.
25. Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
26. J.Smans, B.Jacobs, and F.Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *FMOODS*, pages 220–239, 2008.
27. Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI '10*, pages 63–76, 2010.
28. 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.
29. K. Rustan Leino and Peter Muller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
30. Peter Müller. Modular specification and verification of object-oriented programs. In *Lecture Notes in Computer Science*, volume 2262, 2002.
31. 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.
32. 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.
33. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
34. D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.
35. John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.
36. K. Rustan, M. Leino, and Peter Müller. Object invariants in dynamic contexts. In *In ECOOP*, 2004.

37. 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.