The Modularity and Automation of Object Propositions

Ligia Nistor and Jonathan Aldrich
School of Computer Science, Carnegie Mellon University
{lnistor,aldrich}@cs.cmu.edu

Abstract. Developing a formal verification system that is powerful, modular, and provides good support for automation is challenging. In this paper, we explore the expressiveness, modularity, and automation of our recently proposed object propositions verification system, using two examples. The first example is a spreadsheet of simple cells that add their inputs and share their output with other spreadsheet cells. The second example is a simulator for two queues of jobs, one containing large jobs the other small jobs, along with a producer, consumer, and controller. The examples illustrate challenges for modular reasoning in the presence of shared state (the cells or queues). We specify the examples using object propositions, prove their correctness, and then compare to separation logic specifications with respect to modularity. We also present the translation rules needed for the automatic translation of the object propositions verification methodology into the Boogie intermediate verification language.

1 Introduction

A significant concern in verification research is the ability to reason modularly about programs with state. Recent work has used substructural logics including separation logic (e.g. as in [25]), permissions [8], and Hoare Type Theory [23] as a way of specifying each function in terms of its effect on the state the function reads and writes (i.e. the function’s footprint). Distinguishing the way that state is shared or not shared in a specification is key to achieving the modularity benefits of substructural approaches.

However, sometimes the specifier wants to hide data that is shared between two abstractions. While higher order logic can support specifications that hide sharing [20], higher order logic specifications are more difficult to automate compared to first-order specifications. In recent work, we proposed object propositions [24] as an extension of abstract predicates [25] that allow state to be shared between two objects without revealing this sharing in the objects’ specifications. The implementations of the two objects have a shared fractional permission [8] to access the common data, but this need not be exposed in their external interface. Our technique was designed as an extension to a linear logic with permissions, but we believe that in principle it could be used to extend a separation logic with similar benefits.
In this paper we describe two example programs written using Java code and modularly verified using *object propositions* [24]. We also give the formal translation rules from the language of object propositions to the Boogie intermediate verification language.

A significant focus of this paper is comparing the modularity benefits of object propositions compared to the prior approaches that we extend. Object propositions are interesting in that they combine the modularity benefits of prior approaches based on substructural logics with the benefits of invariant-based verification approaches:

- like separation logic and permissions, but unlike conventional object invariant and ownership-based work (including [21] and [22]), our system allows “ownership transfer” by passing unique permissions (permissions with a fraction of 1) from one reference to another.
- unlike separation logic and permission systems, but like object invariant work and its extensions (for example, the work of Summers and Drossopoulou [28]), we can modify objects without owning them. More broadly, unlike either ownership or separation logic systems, in our 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.

2 Object Propositions in a Nutshell

The object proposition methodology uses abstract predicates [25] to characterise the state of an object, embeds those predicates in a logical framework, and specifies sharing using fractional permissions [8]. 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.

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 [13], 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 \( x \).

A critical part of our work is allowing clients to depend on a property of a shared object. Other methodologies such as Boogie [5] allow a client to depend only on properties of objects that it owns. Our verification technique also allows a client to depend on properties of objects that it does not (exclusively) own. Summers and Drossopoulou’s work [28] accomplishes the same thing, as they do not have the concept of owner. At the same time, they use the classical invariant technique and that is inflexible.

To gain read or write access to the fields of an object, we have to *unpack* it [10]. 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 [24] 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.

At the end of a public method, we pack [10] the keys back into an object proposition and check that object proposition against the method post-condition. A crucial part of our approach is that when we pack an object to a predicate with a fraction less than 1, we have to pack it to the same predicate that was true before the object was unpacked. The restriction is not necessary for a predicate with a fraction of 1: objects that are packed to this kind of predicate can be packed to a different predicate than the one that was true for them before unpacking.

3 Current Approaches

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 [6] developed access permissions, an abstraction that combines typestate and object aliasing information. Their work is similar to ours from a modularity perspective, but is only suited to finite state abstractions, while object propositions can express predicates over integers.

In Boyland’s [8] work a fractional permission means immutability, not sharing, and ensures non-interference of permissions. Similarly, in [7] 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. Our work is more flexible because we allow multiple clients to write to a common resource even if the fractional permission is less than 1, as long as they maintain an invariant on the resource.

Boogie [5] 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. [20] show how to modularly verify programs written using dynamically-generated bidirectional dependency information. Nanevski et al. [23] developed Hoare Type Theory (HTT), which combines a dependently typed, higher-order language with stateful computations. Both have a complexity overhead that leads to difficult automation.
Summers and Drossopoulou [28] introduce Considerate Reasoning, an invariant-based verification technique adopting a relaxed visible-state semantics. While their work is similar to ours in that we both allow a client to depend on properties of objects that it doesn’t (exclusively) own, they use the more restrictive classical invariant technique where all objects of the same class have to satisfy the same invariant.

Separation logic approaches [23], [11], [9], etc. bypass the limitations of invariant-based verification techniques by requiring that each method describe its footprint. Separation logic allows us to reason about how objects’ state changes over time. On the downside, in previous separation logic systems the specification of a method has to reveal the structures of objects that it uses. Our methodology can be seen as an extension to separation logic-style verification (although strictly speaking we provide a core of linear logic), which provides more modularity for some examples. By encoding our verification in Boogie, we have proved that it is amenable to automation.

The work by Kassios and Kritikos [19] is done in a concurrent setting, by using invariants and permissions. Their idea is to extend separation logic by adding backpointer conditions. Although the work of Kassios and Kritikos is done for concurrent programs, their solution is orthogonal to our object propositions that deal with aliasing in a single thread.

There is a body of research describing the encoding of linear logic or fragments of it into first order logic. In his Ph.D. thesis [26] Jason Reed presents an encoding of the entirety of linear logic into first order logic. The major technical difference between Reed’s encoding and ours is that he encodes uninterpreted symbols while our encoding is done inside the theory of object propositions.

Heule et al. [15] present an encoding of abstract predicates and abstraction functions in Boogie. Their encoding is quite different from ours. For example, their exhale operation aggressively havoc the heap and preserves information only for those locations to which the method holds direct or known-folded permission after the exhale.

4 Example I: Cells in a Spreadsheet

We consider the example of a spreadsheet, as described in [20]. In our spreadsheet each cell contains an add formula that adds two integer inputs. Each cell may refer to two other cells. 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 Figure 2, we present the code implementing a cell in a spreadsheet.

4.1 Verification of Cells in a Spreadsheet Using Separation Logic

In conventional first-order separation logic, the specification of any method has to describe the entire footprint of the method, i.e., all heap locations that are being touched through reading or writing in the body of the method. That is,
the shared cells $a3$ and $a6$ have to be specified in the specification of all methods that modify the cells $a1$ and $a2$.

The specification in separation logic exposes some of the 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, we define the following predicate:

$$
\text{SepOK}(\text{cell}) \equiv (\text{cell.in1} \rightarrow x1) \ast (\text{cell.in2} \rightarrow x2) \ast (\text{cell.out} \rightarrow o) \ast (\text{cell.dep1} \rightarrow d1) \ast (\text{cell.dep2} \rightarrow d2) \ast (x1 + x2 == o) \ast (\text{SepOK}(\text{d1.ce}) \land ((\text{d1.ce.in1} \rightarrow o \land \text{d1.input} == 1) \lor (\text{d1.ce.in2} \rightarrow o \land \text{d1.input} == 2))) \ast (\text{SepOK}(\text{d2.ce}) \land ((\text{d2.ce.in1} \rightarrow o \land \text{d2.input} == 1) \lor (\text{d2.ce.in2} \rightarrow o \land \text{d2.input} == 2))).
$$

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 if we want to assert the predicate about two separate nodes whose subtrees overlap due to a DAG structure (e.g. $a1$ and $a2$ in Figure 1). If
class Dependency {
    Cell ce;
    int input;
}

class Cell {
    int in1, in2, out;
    Dependency dep1, dep2;

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

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

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

Fig. 2. Cell class

Fig. 3. Cells in a cycle

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:

\[
Basic(cell) \equiv \exists x_1 : int, x_2 : int, o : int, d : Dependency.(cell.in1 \rightarrow x_1) \times (cell.in2 \rightarrow x_2) \times (cell.out \rightarrow o) \times (cell.dep \rightarrow d). 
\]

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

\[
\{Basic(a2) \times Basic(a5) \times SepOK(a1)\}
\]

\[
a1.setInput1(10);
\]

\[
\{Basic(a2) \times Basic(a5) \times SepOK(a1)\}
\]

\[
\{********** missing step **********\}
\]

\[
\{Basic(a4) \times Basic(a1) \times SepOK(a2)\}
\]

\[
a2.setInput1(20);
\]

In the specification above,

\[
SepOK(a1) \equiv a1.in1 \rightarrow x1 \times a1.in2 \rightarrow x2 \times a1.out \rightarrow o \times x1 + x2 = o \times
(SepOK(a4) \land a4.in1 = o) \times (SepOK(a3) \land a3.in1 = o)
\]

and
\[
SepOK(a2) \equiv a2.in1 \rightarrow z1 \ast a2.in2 \rightarrow z2 \ast a2.out \rightarrow p \ast z1 + z2 = p \ast (SepOK(a3) \land a3.in2 = p) \ast (SepOK(a5) \land a5.in1 = p)
\]

In separation logic, the natural pre- and post-conditions of the method \texttt{setInput1} are \texttt{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. Note that the pre-condition does not need to be of the form \texttt{SepOK(this, carrier)} where \texttt{carrier} is all the cells involved as in Jacobs \textit{et al}.'s work [17]. This is because \texttt{SepOK} is a recursive abstract predicate that states in its definition properties about the cells that depend on the current \texttt{this} cell and thus we do not need to explicitly carry around all the cells involved. The natural specification of \texttt{setInput1} would be \texttt{SepOK(this) ⇒ SepOK(this)}.

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

The specification of \texttt{setInput1} would become
\[
\forall \alpha, \beta, x . \ (SepOK(this) \land SepOK(this) \equiv \alpha \ast SepOK(x) \ast \beta) \Rightarrow (SepOK(this) \land SepOK(this) \equiv \alpha \ast SepOK(x) \ast \beta).
\]

The modification is unnatural: the specification of \texttt{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 \texttt{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 \texttt{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.

Another way of addressing this problem in separation logic is by collecting all the object structure into a mathematical structure and then asserting all the necessary properties using the separation logic star operator. This approach is taken by Jacobs \textit{et al}. [17] for the specification and verification of the composite pattern.

Standard separation logic approaches will have a difficult time trying to modularly 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 [18], which implements this abstraction.
4.2 Verification of Cells in a Spreadsheet Using Object Propositions

In Figure 4, we present the Java class from Figure 2 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, we use linear logic [14] to write the specifications. Pre- and post-conditions are separated with a linear implication $\rightarrow$ 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 [24]. 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. The critical property of an invariant is that it cannot have parameters that represent the fields of the current object. This is because invariants are supposed to hold at the boundaries of methods and if the parameters are changed inside a method, the invariant is broken and other aliases to the same object cannot rely on the invariant anymore. Thus, the invariant for the cell class is $OK()$, while the predicates $In1(int \ x1), In2(int \ x2), OKDep(int \ o)$ cannot be invariants because they have parameters that refer to the fields of the current object and can change.

The predicate $OK()$ in Figure 4 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 for the dependency cells (such as in $c@k\ OK()$, 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.

We have translated this example using object propositions [2] into the Boogie language following the translation rules from Section 7.

5 Example II: 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,
class Dependency {
    Cell ce;
    int input;
    predicate OKdep(int o) ≡ ∃ c, k, i.this.ce → c ⊗ this.input → i ⊗
        ((i = 1 ⊗ c@½ In1(o) ⊗ c@k OK()) ⊕ (i = 2 ⊗ c@½ In2(o) ⊗ c@k OK()))
}

class Cell {
    int in1, in2, out;
    Dependency dep1, dep2;
    predicate In1(int x1) ≡ this.in1 → x1
    predicate In2(int x2) ≡ this.in2 → x2
    predicate OK() ≡ ∃ x1 : int, x2 : int, o : int, d1 : Dependency, d2 : Dependency.
        this@½ In1(x1) ⊗ this@½ In2(x2) ⊗ x1 + x2 = o ⊗ this.out → o ⊗
        this.dep1 → d1 ⊗ this.dep2 → d2 ⊗ d1@1 OKdep(o) ⊗ d2@1 OKdep(o)
    void setInputDep(int newInput) {
        if (this.dep1 != null) {
            if (this.dep1.input == 1) this.dep1.ce.setInput1(newInput);
            else this.dep1.ce.setInput2(newInput);
        }
        if (this.dep2 != null) {
            if (this.dep2.input == 1) this.dep2.ce.setInput1(newInput);
            else this.dep2.ce.setInput2(newInput);
        }
    }
    void setInput1(int x) {
        ∀ k.(this@k OK() ⊗ ∃ y1.this@½ In1(y1) → this@k OK())
        { this.in1 = x;
            this.out = this.in1 + this.in2;
            this.setInputDep(this.out); }
    }
    void setInput2(int x) {
        ∀ k.(this@k OK() ⊗ ∃ y2.this@½ In2(y2) → this@k OK())
        { this.in2 = x;
            this.out = this.in1 + this.in2;
            this.setInputDep(this.out); }
    }
}

Fig. 4. Cell class and OK predicate

yet their specification is not affected by this sharing, then the specification should not
reveal the presence of the sharing.

Unfortunately, some of the most popular reasoning techniques available today —
 principally those based on separation logic [27]— cannot hide sharing,
because the specification of a method must mention the entire memory footprint
that the method accesses. We sustain this claim by giving in this paragraph the
full specification in separation logic for a simulator of queues of jobs. Abstract
predicates are not sufficient to hide the details because, as we have seen in Section 4 for the separation logic specification of the cells in a spreadsheet example, there are cases when the different abstract predicates in the pre-condition have to reveal the exact structure of the cells.

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 [20], but their higher-order nature makes them considerably more complicated.

To further illustrate the modularity issues, we present a relatively realistic example. Figure 5 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. We 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.

![Simulator for queues of jobs](image1)

![Modification of the simulator](image2)

The code in Figures 6, 7 and 8 represents the example from Figure 5.

5.1 Verification of Simulator of Queues of Jobs Using Separation Logic

Now, let's imagine changing the code to reflect the modifications in the right image of Figure 5. The current separation logic approaches do not provide enough
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; }} }

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);} }

public class Consumer {  
    Link startJobs;  
    public Consumer(Link s) {  
        startJobs = s;  
    }
    public void consume()  
    { if (startJobs != null)  
        { startJobs = startJobs.next;}}}

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(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; }} }

    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);} }

modularity. Distefano and Parkinson [12] 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 5 using separation logic.

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

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

The pre-condition for the produce() method is:

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

Note that one can think of the definition $Prod(p)$ below
\[ \text{Prod}(p) \equiv \exists ss, es, sl, el. p.\text{startSmallJobs} \rightarrow ss \ast p.\text{endSmallJobs} \rightarrow es \ast p.\text{startLargeJobs} \rightarrow sl \ast p.\text{endLargeJobs} \rightarrow el \]

to be more modular than \text{Prod}(this, ss, es, sl, el) because the predicate exposes fewer parameters, but then the pre-condition for the \text{produce()} method would become:

\[ \text{Prod}(p) \ast \exists ss, sl. (\text{Listseg}(ss, null, 0, 10) \ast \text{Listseg}(sl, null, 11, 100)) \]

This would remove the connection between the predicates \text{Prod} and \text{Listseg} in the pre-condition and would make the pre-condition much less meaningful. There is also an alternative specification for this example using separation logic and where the \text{ss, es, sl, el} parameters of the \text{Prod} predicate are existentially quantified.

The predicate for the Consumer class is

\[ \text{Cons}(c, s) \equiv c \rightarrow s. \]

The pre-condition for the \text{consume()} method is:

\[ \text{Cons}(c, s) \ast \text{Listseg}(s, null, 0, 10). \]

The predicate for the Control class is:

\[ \text{Ctrl}(ct, p1, p2, c1, c2) \equiv ct.\text{prod1} \rightarrow p1 \ast ct.\text{prod2} \rightarrow p2 \ast ct.\text{cons1} \rightarrow c1 \ast ct.\text{cons2} \rightarrow c2. \]

The pre-condition for \text{makeActive()} is:

\[ \text{Ctrl}(this, p1, p2, c1, c2) \ast \text{Prod}(p1, ss, es, sl, el) \ast \text{Prod}(p2, ss, es, sl, el) \ast \text{Cons}(c1, sl) \ast \text{Cons}(c2, ss) \ast \text{Listseg}(ss, null, 0, 10) \ast \text{Listseg}(sl, null, 11, 100). \]

The lack of modularity will manifest itself when we add the two queues as in the right image of Figure 5.

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

\[ \text{Cons}(c, s1, s2) \equiv c.\text{startJobs}1 \rightarrow s1 \ast c.\text{startJobs}2 \rightarrow s2. \]

The pre-condition for the \text{consume()} method becomes:

\[ \text{Cons}(c, s1, s2) \ast \text{Listseg}(s1, null, 0, 10) \ast \text{Listseg}(s2, null, 0, 10). \]

Although the behavior of the Consumer and Producer classes have not changed, the pre-condition for \text{makeActive()} in class Control does change:

\[ \text{Ctrl}(this, p1, p2, c1, c2) \ast \text{Prod}(p1, ss1, es1, sl1, el1) \ast \text{Prod}(p2, ss2, es2, sl2, el2) \ast \text{Cons}(c1, sl1, sl2) \ast \text{Cons}(c2, ss1, ss2) \ast \text{Listseg}(ss1, null, 0, 10) \ast \text{Listseg}(ss2, null, 0, 10) \ast \text{Listseg}(sl1, null, 11, 100) \ast \text{Listseg}(sl2, null, 11, 100) \]

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.

5.2 Verification of Simulator of Queues of Jobs Using Object Propositions

The code in Figures 10, 11 and 12 represents the code from Figures 6, 7 and 8, 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 \(\text{Range}\) is defined in Figure 9.

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

    predicate Range(int x, int y) ≡
        ∃ v:int, o:Link, k:int.
        val → v ⊗ next → 0
        ⊗ v≥ x ⊗ v ≤ y
        ⊗ [o@k Range(x,y) ⊕ o==null]
}
```

Fig. 9. Link class and Range predicate

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

    predicate BothInRange() ≡
        ∃ o1, o2. startSmallJobs → o1
        ⊗ startLargeJobs → o2
        ⊗ ∃ k1, o1@k1 Range(0,10)
        ⊗ ∃ k2, o2@k2 Range(11,100)

    public void produce() {
        ∃ k.this@k BothInRange() →
        ∃ k.this@k BothInRange()
    }
}
```

Fig. 10. Producer class

```java
public class Consumer {
    Link startJobs;

    predicate ConsumeInRange(int x, int y) ≡
        startJobs → o ⊗ ∃ k@k Range(x,y)

    public void consume() {
        ∀ x:int, y:int.
        ∃ k.this@k ConsumeInRange(x,y)
        → ∃ k.this@k ConsumeInRange(x,y)
    }
}
```

Fig. 11. Consumer class

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

    predicate WorkingSystem() ≡
        prod1 → o1 ⊗ prod2 → o2
        ⊗ cons1 → o3 ⊗ cons2 → o4
        ⊗ ∃ k1, o1@k1 BothInRange()
        ⊗ ∃ k2, o2@k2 in BothInRange()
        ⊗ ∃ k3, o3@k3 in ConsumeInRange(0,10)
        ⊗ ∃ k4, o4@k4 in ConsumeInRange(11,100)

    public void makeActive( int i)
    ∃ k.this@k WorkingSystem() →
    ∃ k.this@k in WorkingSystem()
    }
}
```

Fig. 12. Control class

When changing the code to reflect the modifications in the right image of Figure 5, the internal representation of the predicates changes, but their external
semantics stays the same; the producers produce jobs and they direct them to
the appropriate queue, each consumer accesses only one kind of queue (either
the queue of small jobs or the queue of big jobs), and the controller is still the
manager of the system. The predicate BothInRange() of the Producer class is
exactly the same. The predicate ConsumeInRange(x, y) of the Consumer class
changes to
\[
\text{ConsumeInRange}(x, y) \equiv \exists o_1, o_2, k_1, k_2. \text{startJobs}_1 \rightarrow o_1 @ k_1 \text{Range}(x, y) \otimes o_2 @ k_2 \text{Range}(x, y).
\]

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.

We have translated this example using object propositions [3] into the Boogie
language following the translation rules from Section 7.

6 The Grammar of Object Propositions

\[
\text{Prog} ::= \text{CDecl}
\]
\[
\text{CDecl} ::= \text{class } C \{ \text{FldDecl} \text{ PredDecl} \text{ MthDecl} \}
\]
\[
\text{FldDecl} ::= T f
\]
\[
\text{PredDecl} ::= \text{predicate } Q(T x) \equiv R
\]
\[
\text{MthDecl} ::= T m(T x) \text{MthSpec} \{ e; \text{return } e \}
\]
\[
\text{MthSpec} ::= R \Rightarrow R
\]
\[
R ::= P \mid R \otimes R \mid R \oplus R \mid \exists t R \mid \exists k R \mid \exists k, k \text{ binop } t \Rightarrow R \mid \forall t R \mid \forall k R \mid \forall k, k \text{ binop } t \Rightarrow R \mid t \text{ binop } t \Rightarrow R
\]
\[
P ::= r@k Q(T) \mid \text{unpacked}(r@k Q(T)) \mid r.f \rightarrow x \mid t \text{ binop } t
\]
\[
k ::= \frac{n_1}{n_2} \text{ (where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2)\]
\[
e ::= t \mid r.f \mid r.f = t \mid r. m(T) \mid \text{new } C(T) \mid \text{if } (t) \{ e \} \text{ else } \{ e \} \mid \text{let } x = e \text{ in } e \mid t \text{ binop } t \mid t \& t \mid t \| t \mid ! t
\]
\[
t ::= x \mid n \mid \text{null} \mid \text{true} \mid \text{false}
\]
\[
x ::= r \mid i
\]
\[
\text{binop} ::= + | - | \% = | != | \leq | < | \geq | >
\]
\[
T ::= C \mid \text{int} \mid \text{boolean}
\]

The programming language that we are using is inspired by Featherweight
Java [16], extended to include object propositions. We retained only Java con-
cepts relevant to the core technical contribution of this paper, omitting features
such as inheritance, casting or dynamic dispatch that are important but are
handled by orthogonal techniques.

Above we show the syntax of our simple class-based object-oriented language,
that we call the Oprop 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 ob-
ject 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.

7 Translating Object Propositions into Boogie

Below we present the rules of translation of our Oprop language into the Boogie intermediate verification language. Each non-terminal from the grammar in the previous section has a corresponding rule below.

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

At the start of each Boogie program we declare the type Ref that represents object references. A PredicateType is a type with a finite number of inhabitants: the names of the predicates. A FractionType is a map from a pair [reference r, predicate type QP] to an integer representing the fraction k of the object proposition r@k Q(t). A fraction in the Boogie translation is in the interval [0,100]. The type PackedType is similar to FractionType except that each key points to true if and only if the corresponding object proposition is packed.

trans(Prog) ::= type Ref;
type PredicateTypes;
type FractionType = [Ref, PredicateTypes] int;
type PackedType = [Ref, PredicateTypes] bool;
var packed: PackedType;
var frac: FractionType;
const null: Ref;
trans(ClDecl) trans(e)

A class declaration is made of the field, predicate and method declarations.
trans(ClDecl) ::= trans(FldDecl) trans(PredDecl) trans(MthDecl)

Each field is represented by a map from object references to values.
trans(FldDecl) ::= var f: [Ref]trans(T);
Declarations of parameters in function declarations do not contain the keyword var.

trans(DeclNoVar) ::= f: [Ref]trans(T); | packed: PackedType; | frac: FractionType; | xQPred : [Ref]trans(T);
Declarations of parameters in procedure calls do not contain the keyword `var` or type information:
\[\text{trans(DeclNoTypes)} ::= \text{f} \mid \text{packed} \mid \text{frac} \mid xQPred\]

The declaration of an abstract predicate has several steps. First, `xQPred` is a map representing the current value of the parameter `x` of the predicate `Q`. The function `QPred` is used as a trigger: it helps Boogie choose which axiom to apply during the automatic verification. The first axiom is used for packing the predicate `Q`, while the second one is used for unpacking it.

\[\text{trans(PredDecl)} ::= \text{var xQPred : [Ref]trans(T);}\]
\[\text{const unique QP: PredicateTypes;}\]
\[\text{function QPred(x: trans(T)) returns (bool);}\]
\[\text{axiom (forall this:Ref, DeclNoVar ::}\]
\[\text{(QPred(DeclNoTypes))}\]
\[\text{trans(R) ==> packed[this, QP] \&\& (xQPred[this]==x));}\]
\[\text{axiom (forall this: Ref, DeclNoVar ::}\]
\[\text{(QPred(DeclNoTypes))}\]
\[\text{packed[this, QP] \&\& (xQPred[this]==x) ==> trans(R) );}\]

A method is a procedure in Boogie.
\[\text{trans(MthDecl)} ::=\]
\[\text{procedure m(T x) returns (r:trans(T))}\]
\[\text{trans(MthSpec)}\]
\[\{\text{trans(e); var r := trans(e); return r;}\}

When specifying a method, we have to specify the variables that it modifies, its precondition and its postcondition.
\[\text{trans(MthSpec)} ::= \text{modifies DeclNoTypes;}\]
\[\text{requires trans(R); ensures trans(R);}\]

In the translation of predicates, when fractions are existentially and universally quantified, it means that the value of the map `frac` for the appropriate key is positive.
\[\text{trans(R)} ::= \text{trans(P) \mid trans(R) \&\& trans(R) \mid trans(R) || trans(R) \mid}\]
\[\text{exists t:trans(T):trans(R) \mid}\]
\[\text{(trans(k) > 0) \&\& trans(R) \mid}\]
\[\text{(trans(k) > 0) \&\& trans(k) trans(binop) t ==> trans(R) \mid}\]
\[\text{forall t:trans(T):trans(R) \mid}\]
\[\text{t trans(binop) t ==> trans(R) }\]

An object proposition \(r \& k Q(t)\) is packed when the value of the packed map for \([r, QP]\) is true, the value of `frac[r, QP]` is \(\geq k\), and all parameters `t` have the right values, according to the maps `xQPred`.
\[\text{trans(P)} ::= \text{packed[r, QP] \&\& xQPred[r]==t \&\& frac[r, QP]>=k \mid}\]
\[\text{(packed[r, QP] == false) \&\& xQPred[r]==t \&\& frac[r, QP]>=k \mid}\]
\[\text{f[r]== x \mid t trans(binop) t}\]
A fraction of 100 in Boogie corresponds to a fraction of 1 in Oprop.
\[
\text{trans}(k) ::= \frac{r}{QP}, \text{ where } \frac{r}{QP} = \text{int}\left(\frac{k}{n_1}\right) \times 100 \text{ and } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2
\]

Before packing and unpacking an object proposition, we have to signal to Boogie that we want to perform these operations by assuming the appropriate function that is used as a trigger by Boogie.
\[
\text{trans}(e) ::= t \mid f[r] \mid f[r]:=t \mid \text{call } m(r, t) \mid \\
\text{var } c: \text{Ref}; f[c]:=t; \mid \\
\text{if } (\text{trans}(t)) \{ \text{trans}(e1) \} \text{ else } \{ \text{trans}(e2) \} \mid \\
x := \text{trans}(e1); \text{trans}(e2) \mid t \text{ trans(} \text{binop}) t \mid t \&\& t \mid t \| t \mid ! t \mid \\
\text{assume QPred(DeclNoTypes); packed[this, QP]:=false; trans(e); } \mid \\
\text{assume QPred(DeclNoTypes); assert packed[this, QP]; trans(e); }
\]

A term can be a variable or a constant.
\[
t ::= x \mid n \mid \text{null} \mid \text{true} \mid \text{false}
\]

A variable is a reference to an object or to an integer.
\[
x ::= r \mid i
\]

Boogie does not have the binary operator \%- and we have to simulate it.
\[
\text{trans(binop)} ::= + \mid - \mid \\
\text{function modulo}(x:\text{int}, y:\text{int}) \text{ returns } (\text{int}); \\
\text{axiom } (\forall x, y : (0 \leq x) \&\& (0 < y) \implies (0 \leq \text{modulo}(x, y) ) \&\& (\text{modulo}(x, y) \leq y) ) \&\& \\
((0 < x) \&\& (y < 0) \implies (0 < \text{modulo}(x, y) ) \&\& (\text{modulo}(x, y) < -y) ) \&\& \\
((x < 0) \&\& (0 < y) \implies (-y \leq \text{modulo}(x, y) ) \&\& (\text{modulo}(x, y) < 0) ) \&\& \\
((x < 0) \&\& (y < 0) \implies (y \leq \text{modulo}(x, y) ) \&\& (\text{modulo}(x, y) < 0) ))
\]

The basic types that we use are below, but one can define new ones in Boogie.
\[
\text{trans}(T) ::= \text{Ref} \mid \text{int} \mid \text{bool}
\]

8 Conclusion

We have presented two examples verified using object propositions and compared their verification to a separation logic approach. We have also described the formal translation rules from the object proposition methodology to the Boogie intermediate verification language.

References

7. Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Per-

8. John Boyland. Checking interference with fractional permissions. In *Static Analysis

11. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and

20. Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-