

# Aaree

## A Recipe for Analyzing Object-Oriented Models

Sarfraz Khurshid and Darko Marinov

MIT Laboratory for Computer Science  
Cambridge, MA 02139 USA  
{khurshid,marinov}@lcs.mit.edu

December 11, 2001

**Abstract.** This paper presents Aaree, an extension to the first-order, relational language Alloy. Alloy is suitable for modeling structural properties of software. Alloy models can be analyzed automatically using the Alloy Analyzer. However, Alloy lacks support for directly modeling object-oriented and imperative constructs. Aaree provides support for several of these constructs; this enables intuitive modeling of object-oriented programs. We give Aaree a formal semantics through a translation to Alloy, which bears resemblance to compilation of OO languages. Since Aaree models can be automatically translated to Alloy, they can also be automatically analyzed using the existing Alloy Analyzer. We illustrate the use of Aaree by modeling a part of the Java Collections Framework. We focus on analyzing properties of views in this framework.

## 1 Introduction

Alloy [8] is a first-order, declarative language based on relations. Alloy is suitable for specifying structural properties of software. Alloy *specifications* can be analyzed automatically using the Alloy Analyzer (AA) [7]. Given a finite *scope* for a specification, AA translates it into a propositional formula and uses SAT solving technology to generate *instances* that satisfy the properties expressed in the specification.

The nature of AA makes Alloy particularly suited for interactively exploring program design. Alloy can also be used for modeling program implementations. Currently predominant software design and implementation methodologies are object-oriented (OO). However, Alloy has only a limited support for directly specifying OO design and modeling OO programs.

We recently developed VALloy [15], an extension to Alloy, that provides direct support for modeling *inheritance* similar to the single inheritance of Java. Inheritance is an essential feature of OO languages. It allows a (sub)class to inherit variables and methods from superclass(es). Subclasses can *override* some methods, changing the inherited behavior. The overridden methods are *dynamically dispatched*—the actual function to invoke is selected based on the dynamic types of parameters.

Jackson and Fekete [6] proposed a relational model of heap structure in Alloy. Their approach models state of OO programs and can express sharing and mutual influence between objects. However, the approach does not extend Alloy, and it requires modeling heap at a low level, with a lot of detail; Jackson and Fekete sketched some syntactic sugar to mask detail. Also, their approach does not consider method overriding and dynamic dispatch.

This paper presents Aaree, an extension to VAlloy, that enables intuitive modeling of methods that read and write heap. We give Aaree a formal semantics through a translation to Alloy, which bears resemblance to compilation of OO languages. Since Aaree specifications can be automatically translated to Alloy, they can also be automatically analyzed using the existing AA.

The major design goal for Aaree is to develop a high-level notation for exploring OO design and modeling OO programs, while retaining the analyzability of models. In line with the “micromodularity principle” [8], Aaree does not add to Alloy a direct support for every OO construct, such as Java interfaces and encapsulation.

Aaree adds direct support for the following programming constructs:

- Single inheritance and single dynamic dispatch, as in Java;
- Object allocation;
- State (and mutation);
- Recursion;
- Method sequences.

The translation of state from Aaree to Alloy uses an approach for modeling heap [11] different than the approach proposed by Jackson and Fekete. Additionally, the translation of local variables introduces two more approaches for modeling state in Alloy.

We illustrate Aaree by modeling *views* from the Java Collections Framework (JCF) [20]. In databases, views have been used for over 25 years [3]. They offer the ability to access the same data in different ways. The JCF provides several views on objects that implement data structures, e.g., a subset view on a set or a keyset view on a map. Understanding and using views can be hard, especially in the presence of modifications, several views on the same object, or views on objects that themselves are views. Aaree allows easy modeling of views and automatic analysis of their properties, which can help in formalizing and understanding the semantics of views.

Having an easy way to model commonly used OO constructs is important for several reasons. First, it enables automatic analysis of models of OO programs. Second, it allows developing specifications for these programs. Third, such specifications can be used to test the actual implementations, for example using the TestEra framework [14].

The rest of this paper is organized as follows. Section 2 gives an example that illustrates the key Aaree constructs by building a model of subset view from the JCF. Section 3 defines a semantics for Aaree through a translation to Alloy. Section 4 presents Aaree specifications that model several other views from the JCF. Section 5 reviews related work, and Section 6 presents our conclusions.

## 2 Example

We illustrate Aaree by developing a model of subset views from the Java Collections Framework (JCF). This model involves inheritance and recursive methods. We first model an abstract set of objects similar to `java.util.AbstractSet`. We then model a class that implements this abstract class and determines element membership based on `equals` method. This model uses an Alloy set to represent the elements, instead of using some concrete representation like trees or hash tables. Next, we model a set that represents a subset view on another set. We also show how a first-class function can be modeled. Finally, we illustrate how behavior of sequences of methods can be automatically analyzed.

Following Alloy [5], an Aaree specification consists of a sequence of paragraphs. A paragraph either introduces a class of objects or defines methods on these classes. These methods are specified using Alloy-like formulas. These formulas are in relational, first-order logic extended with transitive closure.

Aaree methods are either *pure* or *general*. A pure method expresses properties of one state. A general method relates a pre-state (i.e., the state immediately prior to the method invocation) to a post-state (i.e., the state immediately after the method invocation). We explain Aaree and Alloy notations as we introduce them.

### 2.1 Abstract set

The following Aaree specification models an abstract class with two methods:

```
abstract class Set {}

method Set::isSubset(s: Set) {
  all o: Object | contains(o) => s..contains(o) }

method Set::equals(s: Object) {
  s instanceof Set
  isSubset(s)
  s..isSubset(this) }
```

`Set` is an abstract class that, by default, extends the class `Object`. The method `isSubset`, introduced by the keyword `method`, models the (pure) boolean method that determines whether a set is a subset of another. This method has two arguments: `s` and the implicit `this` argument introduced with `':'`. The method body specifies that every object contained in `this` is also contained in `s`; `'.'` denotes set membership, `'=>'` denotes implication, and `'..'` denotes method invocation. If no receiver object is specified for a method invocation, `this` is assumed to be the receiver.

The (pure) method `equals` determines whether the set `this` is equal to the input set `s`. The method body is a formula that is (implicitly) a conjunction of three sub-formulas: the first sub-formula requires `s` to be an object of class `Set` or a subclass of `Set` (`instanceof` models `instanceof` in Java); the second and third sub-formulas require `this` to be a subset of `s` and vice versa.

## 2.2 Concrete set

The following Aaree specification models an implementation of the abstract class `Set` and defines some methods on it:

```
class SetImpl extends Set {
  s: set Object }

method SetImpl::SetImpl() {
  no s'
  modifies: s }

method SetImpl::repOk() {
  all disj e1, e2: s | !e1..equals(e2) }

method SetImpl::contains(o: Object) {
  some e: s | o..equals(e) }

method SetImpl::add(o: Object) {
  s..contains(o) => s' = s, s' = s + o
  modifies: s }

method SetImpl::remove(o: Object) {
  s' = s - { e: s | e..equals(o) }
  modifies: s }
```

Every object of `SetImpl` has a field `s` that denotes a (mathematical) set of objects; `set` is an Alloy keyword. `SetImpl` has a constructor that creates an empty set. This constructor is a (general) method that modifies state; this method denotes a relation between a pre- and a post-state. The prime ‘`'`’ represents the value of a field in the post-state. The quantifier `no` specifies that `s` is empty in the post-state. The Aaree keyword `modifies` lists the fields modified by this method. All other fields retain their pre-state values.

The method `repOk` expresses the structural correctness of a `SetImpl` object. It requires that for all distinct objects `e1` and `e2` in `s`, `e1` is not `equals` of `e2`. The keyword `disj` declares disjoint (singleton) subsets, and ‘`!`’ denotes negation.

The method `contains` specifies that the `SetImpl` object `this` contains the object `o` if there exists an object in `s` that is `equals` of `o`.

The method `add` adds an object `o` to a set. If `s` already contains `o` in the pre-state, `s` remains the same; otherwise, `s` in the post-state is union of `s` in the pre-state and `o`. The operator ‘`+`’ denotes union, and ‘`,`’ denotes the else-branch of a condition. This method modifies the field `s`. The method `remove` uses set comprehension to specify removal of an object from a set.

## 2.3 Subset view

We next model a subset view in Aaree. In JCF, the class `java.util.TreeSet` provides an implementation of a (sorted) set and also of a subset view. A subset view is created by invoking `s.subSet(fromElement, toElement)`, where `s` is the *backing* set, and the result is a portion of `s` with elements ranging from `fromElement`, inclusive, to `toElement`, exclusive. The returned set is backed by `s`, so changes in the view are reflected in `s`, and vice-versa. Note that a view can also be a view on another subset view.

In general, a subset view on a set can be created by giving a *predicate* (boolean returning function) that determines membership in the view. In a language supporting first-class functions, a view analogous to the above view in JCF could be created by `s.subSet(p)`, where `s` is a set and `p` is a predicate defined, for example, as `(lambda (o) (and (>= o fromElement) (< o toElement)))`.

The following Aaree specification models predicates:

```
class Predicate {
  def: set Object }

method Predicate::Predicate(d: set Object) {
  def' = d
  modifies: def }

method Predicate::repOk() {}

method Predicate::admits(o: Object) {
  o in def }
```

A predicate object has a field `def` that denotes a (mathematical) set of objects that defines this predicate. This field stores the *admissible* objects, i.e., the objects for which the functional predicate evaluates to true. The constructor `Predicate` creates a new predicate object given a set of admissible objects. The representation invariant on predicate objects is always true. The method `admits` determines whether the input object `o` is admissible for this predicate; ‘`in`’ denotes subset/membership operation.

The following Aaree specification models a subset view:

```
class Subset extends Set {
  on: Set,
  filter: Predicate }

method Subset::Subset(s: Set, p: Predicate) {
  init'(s,p)
  modifies: on + filter }

method Subset::init(s: Set, p: Predicate) {
  on = s
  filter = p }

method Subset::repOk() {
  all ss: *on | ss !in ss.^on } // acyclic

method Subset::contains(o: Object) {
  filter..admits(o) && on..contains(o) }

method Subset::add(o: Object) {
  filter..admits(o) => on..add(o), modifies: }

method Subset::remove(o: Object) {
  filter..admits(o) => on..remove(o), modifies: }
```

Every object of `Subset` has two fields: `on` that refers to the set object that backs this subset view and `filter` that is a predicate that determines membership for this subset view.

`Subset` has a constructor that takes a backing set and a membership predicate. This constructor invokes the method `init` that sets the fields `on` and `filter` of `this` to the parameters `s` and `p`. Notice that `init` is a pure method. The invocation

`init'(s,p)` indicates that this method should be evaluated in post-state, not pre-state.

The method `repOk` specifies that a subset view is structurally valid if there are no cycles following on fields, i.e., for each subset view, there is (transitively) a backing set that is not a view. The operators `'^'` and `'*'` denote transitive closure and reflexive transitive closure, respectively.

The method `contains` specifies the membership of an object in a subset view. It requires that the input object `o` is both admissible by the predicate `filter` and also in the backing set of `this`.

The method `add` first checks if `o` is admissible by `filter`: if so, `add` simply invokes the method `add` for the backing set; otherwise, `add` does not modify any field(s). The method `remove` is similar.

The abstract class `Set` also specifies a method to create a new subset view on a set object. The following Aaree method models `getSubset` that returns a view defined by predicate `p`:

```
method Set::getSubset(p: Predicate): Subset {
  result = new Subset(this, p) }
```

The Alloy keyword `result` represents the return value of a method.

## 2.4 Analysis

We next present an interactive analysis of some properties of the above Aaree specification. We analyze Aaree specifications by translating them to Alloy and using the Alloy Analyzer. We describe in this section only the result of the analysis, and we describe the translations in the next section.

Consider the following Aaree sequence for checking addition:

```
check AddOk1 {
  all si: SetImpl, u: Subset, o: Object {
    si.add(o);
    assert { u.contains(o) }
  }
}
```

The Aaree keyword `check` states a property that should be checked. The operator `';`' introduces method sequencing. The assertion stated with `assert` expresses a property that should hold at a certain control point in the sequence. The above property states that an addition of an object `o` to a set `si` should also make `o` a member of a subset `u`.

The Alloy Analyzer checks (the translation into Alloy of) `AddOk1` and produces a counterexample, as expected, because `u` is not constrained to be a subset view on `si`. We next add this constraint:

```
check AddOk2 {
  all si: SetImpl, u: Subset, o: Object {
    u.on = si
    si.add(o);
    assert { u.contains(o) }
  }
}
```

The Alloy Analyzer checks `AddOk2` and once again produces a counterexample. This time, the assertion fails because the predicate of `u` does not admit `o`. We next add this constraint:

```

check AddOk3 {
  all si: SetImpl, u: Subset, o: Object {
    u.filter..admits(o)
    u.on = si
    si..add(o);
    assert { u..contains(o) }
  }
}

```

The Alloy Analyzer checks `AddOk3` and once again produces a counterexample. This time, the assertion fails because `si` contains itself as an element. In this case, invoking `si.contains(si)` is a problem—in Java, it results in an infinite recursion; in Aaree, it results in an underspecified relation as we explain in Section 3.4. We rule out such instances by modifying `repOk` for `SetImpl`:

```

method SetImpl::repOk() {
  all disj e1, e2: s | !e1..equals(e2)
  this !in ~s }

```

The Alloy Analyzer again checks `AddOk3` and once again produces a counterexample. This time, the assertion fails because `si` contains as an element a subset view on `si`, i.e., `si` indirectly contains itself, which leads to the above problem. A way to rule out these instances is to constrain sets not to include sets as members. We introduce `repOk` for `Set` and invoke it from `repOk()`s for `SetImpl` and `Subset`:

```

method Set::repOk() {
  all s: Set | !contains(s) }

method SetImpl::repOk() {
  super..repOk()
  all disj e1, e2: s | !e1..equals(e2) }

method Subset::repOk() {
  super..repOk()
  all ss: *on | ss !in ss.^on } // acyclic

```

With this correction in place, the Alloy Analyzer checks `AddOk3` and this time reports no counterexamples.

### 3 Aaree

In this section, we define the semantics of Aaree through a translation into Alloy, whose formal semantics can be found in [5]. Aaree adds to Alloy constructs for modeling object-oriented and imperative programs. In particular, Aaree adds the following keywords:

- `class` declares a new class;
- `abstract` specifies that a class is abstract;
- `new` corresponds to object creation;
- `instanceof` checks subclass relationship;
- `method` declares a new method;
- `modifies` specifies the part of state that a method modifies;
- `local` declares local variables used in method sequences;
- `inline` declares that a method invocation does not use modifies from the invoked method.

Also, Aaree allows methods to refer to the post-state using primed names (‘’). These constructs are added in a syntactically obvious fashion and illustrated through examples in this paper. The full grammar of Alloy can be found in [5].

We organize the presentation of our translation by the constructs that Aaree directly supports. We illustrate each construct on (a part of) the example from Section 2.

### 3.1 Inheritance

The translation for inheritance is similar to compilation of OO languages, involving creation of virtual function tables. Details can be found in [15]. In outline, the translation has six steps:

1. Compute a hierarchy of class declarations.
2. Construct `sig Class` and `sig Object`.
3. Change each `class` declaration into `disj sig` declaration.
4. Rename uniquely each method.
5. Add dispatching functions.
6. Replace `super` with appropriate static invocation.

We illustrate these steps by translating the `equals` methods (with their overriding) presented in Section 2.

Step 1 computes the following class hierarchy:

```
Object
+-- Set
    +-- SetImpl
    +-- Subset
+-- Predicate
```

Step 2 constructs the signatures `Class` and `Object`:

```
sig Class {}

static disj sig Object_Class, SetImpl_Class, Subset_Class, Predicate_Class extends Class

sig Object {
  class: Class }

fact ObjectClasses {
  (Object - Set - Predicate).class in Object_Class
  no Set - SetImpl - Subset // abstract class
  SetImpl.class in SetImpl_Class
  Subset.class in Subset_Class
  Predicate.class in Predicate_Class }

fun Object::Object_equals(o: Object) {
  this = o }
```

This step declares a disjoint static signature for each concrete class in the Aaree specification. Each atom of signature `Object` has a field `class` that denotes the dynamic type of this object. The values for this field are set by the *fact* `ObjectClasses`.

This step also adds for `Object` the default `equals` method that compares object identities (as in Java). This step renames the function to `Object_equals` since Alloy functions must have unique names.

Step 3 changes each class declaration into a `disj sig` declaration and adds `extends Object` where needed.

Step 4 renames the `equals` in class `Set`:

```
fun Set::Set_equals(s: Object) {
  ... }
```

This step does not translate method bodies.

Step 5 adds the dispatching function for `equals`:

```
fun Object::equals(o: Object) {
  This.class = Object_Class => This..Object_equals(o)
  This.class = SetImpl_Class => This..Set_equals(o)
  This.class = Subset_Class => This..Set_equals(o)
  This.class = Predicate_Class => This..Object_equals(o)
}
```

Since `Predicate` does not override `equals`, it inherits it from `Object`; likewise, `SetImpl` and `Subset` inherit `equals` from `Set`.

Step 6 uses the class hierarchy information to replace invocations on `super` with appropriate static invocations; this step does not translate anything in our running example.

### 3.2 State

To allow sequencing of methods, our translation introduces a model of state in Alloy. We adopt a relational model of the state/heap where fields of objects are treated as relations among objects. A valuation of these relations defines a state; different valuations give rise to different states.

The translation first introduces a new signature `State` and then translates (pure and general) methods that express properties on state(s).

For our running example, the signature `State` is:

```
sig State {
  // fields
  s: SetImpl -> Object,
  on: Subset ->! Set,
  filter: Subset ->! Predicate,
  def: Predicate -> Object }
```

Each atom of `State` models a state. Each relation in `State` corresponds to one of the four field declarations in the Aaree specification—the translation essentially flattens out the hierarchical structure of the Aaree specification.

Our approach to modeling state differs from the approach proposed by Jackson and Fekete [6] in that their approach does not have a translation step, but keeps the state hierarchical. Their approach models mutation by explicitly modeling references. We have yet to evaluate which approach leads to faster analysis.

### 3.3 Methods

Aaree methods are translated into Alloy functions. This translation involves translation of method declarations and method bodies.

We first illustrate translation of a pure method, using the following example:

```
method SetImpl::contains(o: Object) {
  some e: s | o..equals(e) }
```

The translation generates the following Alloy function:

```
fun SetImpl::SetImpl_contains(t: State, o: Object) {
  some e: t.s[this] | o..equals(t, e) }
```

This function has, in addition to the method parameter, one **State** parameter. This parameter is also added to each invocation in the method body. (Invocations in a pure method can invoke only pure, and not general, methods.) Also, each field access in the body, in this example `this.s`, is replaced with the access of the corresponding value in the state, i.e., `t.s[this]`.

We next illustrate translation of a general method, using the following example:

```
method SetImpl::add(o: Object) {
  s..contains(o) => s' = s, s' = s + o
  modifies: s }
```

The translation generates the following Alloy function:

```
fun SetImpl::SetImpl_add(t, t': State, o: Object) {
  t.s[this]..contains(t, o) => t'.s[this] = t.s[this],
  t'.s[this] = t.s[this] + o
  modifiesFields_s(t, t', this)
  modifies(t, t', this) }
```

This function has, in addition to the method parameter, two **State** parameters, for pre-state and post-state. We use primed notation in `t'` to signify post-state; `'` has no semantic significance in Alloy. The field accesses and the (pure) method invocation in the body of this general method are translated as illustrated earlier.

The `modifies` clause, which specifies the modified fields, is translated into several function invocations. For each set of fields that is modified, in this example `s`, an appropriate `modifiesFields` function is invoked with the set of objects, in this example `this`, that have exactly those fields modified, i.e., `modifiesFields_s(t, t', this)`. (The invocations of `modifiesFields` should be on disjoint sets of objects.) Also, the function `modifies` is invoked with all the objects that are modified. This function specifies that the fields of all other objects remain unchanged:

```
fun modifies(t, t': State, so: set Object) {
  all o: Object - so {
    t.s[o] = t'.s[o]
    t.on[o] = t'.on[o]
    t.filter[o] = t'.filter[o]
    t.def[o] = t'.def[o]
  } }
```

The function `modifiesFields_s` is similar to `modifies`, except that it does not require `s` to remain unchanged.

In general, the translation adds state(s) to the expressions that can appear in the body of a method in the following way:

- Replace every invocation `o..pm(...)` of a pure method `pm` with `o..pm(t, ...)`, where `t` is (pre-)state, and every invocation `o..pm'(...)` with `o..pm(t', ...)`, where `t'` is post-state.
- Replace every invocation `o..gm(...)` of a general method `gm` with `o..gm(t, t', ...)`.
- Replace every field access `o.f` with `t.f[o]`.
- Replace every field access `o.f'` with `t'.f[o]`.

### 3.4 Recursion

Alloy Analyzer currently does not support recursive functions. We allow Aaree specification to contain recursive methods. The translation of methods given above results in recursive functions. We next outline the translation that eliminates recursive functions so that we can use the current Alloy Analyzer.

Elimination of recursion introduces new relations in the signature `State` and constrains these relations using the corresponding method definitions. For each recursive method, the translation introduces a new relation; in our running example:

```
sig State {
  // fields
  ... // as above

  // recursion
  equalsR: Object -> Object,
  containsR: Object -> Object,
  addR: State -> Object -> Object,
  removeR: State -> Object -> Object }
```

Since `add` and `remove` are general methods, the types of corresponding relations have an additional (post-)state.

The translation eliminates recursion from `equals` by replacing it with:

```
fact EqualsFixPoint { all This, o: Object | all t: State |
  This..equals(t, o)
  <=>
  {
    This.class = Object_Class => This..Object_equals(t, o)
    This.class = SetImpl_Class => This..Set_equals(t, o)
    This.class = Subset_Class => This..Set_equals(t, o)
    This.class = Predicate_Class => This..Object_equals(t, o)
  } }

fun Object::equals(t: State, o: Object) {
  o in equalsR[t][this] }
```

The fact `EqualsFixPoint` evaluates the relation `equalsR` to a fixed point of `equals` methods. If the definition of a recursive method has several fixed points, AA will explore all of them.

The translation eliminates recursion from general methods in a similar fashion; for the `add` method:

```
fact AddFixPoint { all t, t': State | all This, o: Object |
  This..add(t, t', o)
  <=>
  {
    This.class != Object_Class
    This.class = SetImpl_Class => This..SetImpl_add(t, t', o)
    This.class = Subset_Class => This..Subset_add(t, t', o)
    This.class != Predicate_Class
  } }

fun Object::add(t, t': State, o: Object) {
  o in addR[t][t'][this] }
```

### 3.5 Object creation

Aaree supports object creation (in general methods). The translation adds a field `free: set Object` to the `State` signature; for each state `t`, `t.free` is the

set of unallocated objects in that state. Object creation allocates an object by removing it from the free-set.

The translation disallows the objects in `t.free` to be pointed to along some field. For expediting the analysis, the translation also disallows free objects to point to any object. For our running example, the translation adds:

```
fact Free { all t: State | with t {
  no free.(s + on + filter + def)
  no free.^(s + on + filter + def)
} }
```

It also adds the following constraint to the body of `modifies`:

```
t'.free = t.free - so
```

In our running example, the constructor for `Subset` is translated to the following function:

```
fun Subset_new(t, t': State, s: Set, p: Predicate): Subset {
  result in t.free
  result.class = Subset_Class
  result.Subset_init(t', s, p)
  modifiesFields_on_filter(t, t', result)
  modifies(t, t', result) }

fun Subset::Subset_init(t: State, s: Set, p: Predicate) {
  t.on[this] = s
  t.filter[this] = p }
```

The function `Subset_new` requires `result`, i.e., the freshly allocated object, to be unallocated in the pre-state and also appropriately constrains its class. `Subset_new` invokes the helper function `Subset_init` that assigns, in the post-state, the values to the fields of `result`. `Subset_new` also contains the translated `modifies` clause.

### 3.6 Sequences

Aaree supports analysis of sequences of methods invocations. Aaree provides notation for expressing sequences and making assertions that are to be checked at different control points.

Consider, for example, the following sequence that involves a conditional branch:

```
check IfExample {
  all si: SetImpl, u: Subset, p: Predicate, o: Object {
    if (p..admits(o)) {
      u..remove(o);
      assert { !si..contains(o) }
    }
    si..add(o);
    assert { p..admits(o) => u..contains(o) }
  }
}
```

The translation generates an Alloy assertion that threads state through the sequence:

```
static part sig t0, t1, t2 extends State {}

assert IfExample {
  all si: SetImpl, u: Subset, p: Predicate, o: Object {
    if (p..admits(t0, o)) {
      u..remove(t0, t1, o)
    }
  }
}
```

```

    } else {
      t1 = t0
    }
    si..add(t1, t2, o)
  } => {
    !si..contains(t1, o)
    p..admits(t2, o) => u..contains(t2, o)
  }
}

```

### 3.7 Local variables

Aaree also supports the use of local variables in the analysis of method sequences. The following example illustrates declaration and use of local variables:

```

check Addition {
  all o: Object {
    local si: SetImpl, p: Predicate, u: Subset;
    si = new SetImpl();
    p = new Predicate(Object);
    u = si..getSubset(p);
    si..add(o);
    si = new SetImpl();
    si..remove(o);
    assert { u..contains(o) }
  }
}

```

Note that Aaree allows combining quantified variables (*o* in the example) and local variables (*si*, *p*, and *u* in the example); the difference is that the values of local variables depend on the state (in the example, only *si* changes its value). It is possible to translate local variables to state as presented above in modeling state. We next present two more approaches that the translation can use for local variables.

The *functional* approach adapts McCarthy's rules for representing memory writes and reads in logic [16]:

```

sig LocalVar {}
static part sig LV_si, LV_p, LV_u extends LocalVar {}

sig State {
  ... // as above
  localMem: LocalMem }
sig LocalMem {
  mem: LocalMem,
  var: LocalVar,
  obj: Object }
static sig LocalMem0 extends LocalMem {}
fact { LocalMem0.m = LocalMem0 }

fun localWrite(t, t': State, l: LocalVar, o: Object) {
  with t'.localMem { mem = t.localMem var = l obj = o } }
fun localRead(t: State, l: LocalVar): Object {
  some ml: t.localMem.*mem { // ml is the latest update of l
    ml.var = l
    no mp: t.localMem.*mem - ml.*mem | mp.var = l
    result = ml.obj } }

```

This approach introduces `sig LocalVar` and adds to the state a field for local memory. An atom of `sig LocalMem` models one write to the local memory: the value `obj` is written to the local variable `var` in the memory `mem`. The memory `LocalMem0` represents the initial local memory. The functions `localWrite` and `localRead` relate state(s) with the local variable and the value written/read.

Using the functional approach, the above sequence can be translated to the following:

```
static part sig t0, t1, t2, t3, t4, t5, t6 extends State {}

assert Addition {
  all o: Object {
    t0.localMem = LocalMem0
    writeLocal(t0, t1, LV_si, SetImpl_new(t0, t1))
    writeLocal(t1, t2, LV_p, Predicate_new(t1, t2, Object))
    writeLocal(t2, t3, LV_u,
      readLocal(t2, LV_si)..getSubset(t2, t3, read(t2, LV_p)))
    readLocal(t3, LV_si)..add(t3, t4, o) && t3.localMem = t4.localMem
    writeLocal(t4, t5, LV_si, SetImpl_new(t4, t5))
    readLocal(t5, LV_si)..remove(t5, t6, o) && t5.localMem = t6.localMem
  } => {
    readLocal(t4, LV_u)..contains(t4, o)
  }
}
```

An alternative approach for translating local variables (in sequences that have no loops) is to rename local variables to the static single assignment (SSA) form [1] and replace them with (universally) quantified variables. For the above example, applying this translation gives the following Alloy code:

```
static part sig t0, t1, t2, t3, t4, t5, t6 extends State {}

assert Addition {
  all o: Object {
    all si_0: SetImpl, p_0: Predicate, u_0: Subset, si_1: SetImpl {
      si_0 = SetImpl_new(t0, t1)
      p_0 = Predicate_new(t1, t2, Object)
      u_0 = si_0..getSubset(t2, t3, p_0)
      si_0..add(t3, t4, o)
      si_1 = SetImpl_new(t4, t5)
      si_0..add(t5, t6, o)
    }
  } => {
    u_0..contains(t4, o)
  }
}
```

## 4 Collections

In this section, we develop an Aaree model for views on maps, such as those in Java, e.g., the `keySet` and `entrySet` views on `java.util.TreeMap`. We build on the model presented in Section 2.

A map can be considered as a set of *entries*, where each entry is a *key-value* pair. The following Aaree specification models an abstract map and defines a method for comparing maps for equality:

```
abstract class Map {}

method Map::equals(m: Object) {
  m instanceof Map
  entrySet()..equals(m..entrySet()) }


```

The method `equals` invokes `entrySet`, which returns a set of entries in the map.

The following class `MapImpl` models a concrete map:

```
class MapImpl extends Map {
  keys: SetImpl,
  map: keys.s ->! Object }


```

```

method MapImpl::MapImpl() {
  keys' = inline new SetImpl()
  no map'
  modifies: keys + keys.s + map }

method MapImpl::repOk() {}

method MapImpl::keySet(): Set {
  result = new KeySet(this) }

method MapImpl::entrySet(): Set {
  result = new EntrySet(this) }

method MapImpl::get(k: Object): Object {
  keys..contains(k) =>
  some e: keys.s | e..equals(k) && result = e.map }

method MapImpl::put(k, o: Object) {
  keys'.s' = (keys.s - { e: keys | e..equals(k) }) + k
  map' = (map - { e: keys | e..equals(k) } -> Object) + k -> o
  modifies: keys.s + map }

method MapImpl::remove(k: Object) {
  inline keys..remove(k)
  map' = map - { e: keys | e..equals(k) } -> Object
  modifies: keys.s + map }

```

The two fields `keys` and `map` represent, respectively, the set of keys and the mapping function from keys to values for the concrete map.

The constructor for `MapImpl` invokes the constructor for `SetImpl`. However, the keyword `inline` indicates that this invocation should not use the `modifies` expression for the constructor of `SetImpl`. If the `modifies` expression for the constructor of `SetImpl` was used, it would contradict the `modifies` expression for the constructor of `MapImpl`, and the latter constructor would be inconsistent.

The methods `keySet` and `entrySet` return, respectively, the set of keys and the set of entries in the map.

The method `get` returns the value corresponding to the input key, if such a key exists in the map; otherwise, the output is left unspecified. (Since Alloy is a relational language, non-determinism comes for free.) We can constrain `get` to be deterministic, e.g., to return an explicit `Null` object, if the key is not in the map. The method `put` adds the input key-value pair to the map. The method `remove` simply removes the key-value pair corresponding to the input key from the map.

The class `EntrySet` models a *entry-set* view on a map:

```

class EntrySet extends Set {
  on: Map }

method EntrySet::EntrySet(m: Map) {
  on' = m
  modifies: on }

method EntrySet::repOk() {}

method EntrySet::elements(): set Object {
  all o: MapEntry |
  o in result <=> o.key -> o.value in on.map }

method EntrySet::remove(e: Object) {
  e instanceof MapEntry && on..get(e.key)..equals(e.value)

```

```

=> on..remove(e.key), modifies: }

method EntrySet::add(e: Object) {
  e instanceof MapEntry => on..put(e.key, e.value), modifies: }

```

The elements of an entry-set are exactly the entries in the backing map. Removing an entry from an entry-set removes it from the map. Similarly, adding an entry to an entry-set adds it to the backing map.

The class `MapEntry` models an entry in a map:

```

class MapEntry {
  key: Object,
  value: Object }

method MapEntry::MapEntry(k: Object, v: Value) {
  key' = k
  value' = v
  modifies: key + value }

method MapEntry::repOK() {}

```

An entry is simply a pair of a key and a value.

The class `KeySet` models a *key-set* view on a map:

```

class KeySet extends Set {
  on: Map }

method KeySet::KeySet(m: Map) {
  on' = m
  modifies: on }

method KeySet::repOK() {}

method KeySet::elements(): set Object {
  result = on.keys.s }

method KeySet::remove(e: Object) {
  on..remove(e) }

```

The elements of a key-set are exactly the keys in the backing map. Removing an element from a key-set removes the corresponding key-value pair from the backing map. JCF does not allow additions to a key-set, because it is not possible to specify the value that should be added with the key.

## 5 Related work

Jackson and Fekete [6] proposed an approach for modeling in Alloy object interactions, like those in Java. Their approach models heap using explicit references and captures properties of object sharing and aliasing. However, the approach does not handle inheritance in the presence of method overriding and dynamic dispatch.

We recently proposed VALloy [15], an extension to Alloy, that adds direct support for modeling method overriding and dynamic dispatch similar to the single inheritance of Java. Aaree extends VALloy by adding support for state and mutation, local variables, recursion, and object allocation. Aaree uses an approach [11] for modeling heap different than the approach proposed by Jackson and Fekete.

Alloy has been used to check properties of programs that manipulate dynamic data structures. Jackson and Vaziri [9] developed a technique for analyzing bounded segments of procedures that manipulate linked lists. Their technique automatically builds an Alloy model of computation and checks it against a specification. They consider a small subset of Java, without dynamic dispatch.

We developed TestEra [14], a framework for automated testing of Java programs. In TestEra, specifications are written in Alloy and the Alloy Analyzer is used to provide automatic test case generation and correctness evaluation of programs. Writing specifications for Java collections requires support for advanced programming constructs such as inheritance and recursion. This led us to tackle providing direct support for such constructs in Alloy. Aaree presents some ideas toward that goal.

The Java Modeling Language (JML) [12] is a popular specification language for Java. JML assertions use Java syntax and semantics, with some additional constructs, most notably for quantification. Leveraging on Java, JML specifications can obviously express dynamic dispatch. However, JML lacks static tools for automatic verification of such specifications.

The LOOP project [21] models inheritance in higher order logic to reason about Java classes. Java classes and their JML specifications are compiled into logical theories in higher order logic. A theorem prover is used to verify the desired properties. This framework has been used to verify that the methods of `java.util.Vector` maintain the safety property that the actual size of a vector is less than or equal to its capacity [4].

Object-oriented paradigm has been integrated into many existing languages, typically to make reuse easier. For example, Object-Z [18] extends the Z specification language [19], which enables building specifications in an object-oriented style. Object-Z retains the syntax and semantics of Z, adding new constructs. The major new construct is the class schema that captures the object-oriented notion of a class.. Object-Z allows inheritance to be modeled, but it lacks tool support for automatically analyzing specifications.

Objects and inheritance have also been added to declarative languages. For example, Prolog++ [17] extends Prolog. OOLP+ [2] aims to integrate object-oriented paradigm with logic programming by translating OOLP+ code into Prolog without meta-interpretation.

Keidar et al. [10] add inheritance to the IOA language [13] for modeling state machines, which enables reusing simulation proofs between state machines. This approach allows only a limited form of inheritance, subclassing for extension: subclasses can add new methods and *specialize* inherited methods, but they cannot override those inherited methods, changing their behavior arbitrarily. VALloy allows subclasses to arbitrarily change the behavior of inherited methods.

## 6 Conclusions

In this paper, we described Aaree, an extension to the first-order, relational language Alloy. Alloy has an automatic analyzer, which makes it particularly

sued for interactively exploring program design. However, Alloy has only a limited support for directly specifying OO design and modeling OO programs. In particular, Alloy lacks direct support for modeling inheritance, state and mutations, recursion, local variables, and sequencing of operations; Aaree supports all of these constructs. Thus, Aaree enables intuitive modeling of methods that read and write heap.

We gave Aaree a formal semantics through a translation to Alloy. This translation bears resemblance to compilation of OO languages. Since Aaree specifications can be automatically translated to Alloy, they can also be automatically analyzed using the existing Alloy Analyzer.

We illustrated Aaree by modeling views from the Java Collections Framework. Understanding and using views can be hard, especially in the presence of modifications, several views on the same object, or views on objects that themselves are views. Aaree allows modeling views and analyzing their properties, which can help in formalizing and understanding the semantics of views.

We believe that Aaree can be effectively used for building specifications and checking implementations of Java classes, as well as for checking properties of models of OO programs.

## References

1. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310, White Plains, N.Y., June 1990.
2. Mukesh Dalal and Dipayan Gangopahyay. OOLP: A translation approach to object-oriented logic programming. In *Proc. First International Conference on Deductive and Object-Oriented Databases (DOOD-89)*, pages 555–568, Kyoto, Japan, December 1989.
3. C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Publishing Company, 1995.
4. Marieke Huisman, Bart Jacobs, and Joachim van den Berg. A case study in class library verification: Java's Vector class. *Software Tools for Technology Transfer*, 2001. (to appear).
5. Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. Available online: <http://sdg.lcs.mit.edu/alloy/book.pdf>.
6. Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.
7. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.
8. Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Vienna, Austria, September 2001.
9. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, August 2000.

10. Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, pages 478–487, Limerick, Ireland, June 2000.
11. Sarfraz Khurshid and Darko Marinov. TALloy: Multiple instances in a single Alloy model. (unpublished extended abstract), May 2001.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001).
13. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
14. Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.
15. Darko Marinov and Sarfraz Khurshid. VALloy: Virtual functions meet a relational language. (submitted for publication), October 2001.
16. John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.
17. Chris Moss. *Prolog++ The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.
18. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
19. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
20. Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. <http://java.sun.com/j2se/1.3/docs/api/>.
21. Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Proc. Tools and Algorithms for the Construction and Analysis of Software (TACAS), (Springer LNCS 2031, 2001)*, pages 299–312, Genoa, Italy, April 2001.