VAlloy – Virtual Functions Meet a Relational Language

Darko Marinov and Sarfraz Khurshid

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

Abstract. We propose VAlloy, a veneer onto the first order, relational language Alloy. Alloy is suitable for modeling structural properties of object-oriented software. However, Alloy lacks support for dynamic dispatch, i.e., function invocation based on actual parameter types. VAlloy introduces virtual functions in Alloy, which enables intuitive modeling of inheritance. Models in VAlloy are automatically translated into Alloy and can be automatically checked using the existing Alloy Analyzer. We illustrate the use of VAlloy by modeling object equality, such as in Java. We also give specifications for a part of the Java Collections Framework.

1 Introduction

Object-oriented design and object-oriented programming have become predominant software methodologies. An essential feature of object-oriented languages is *inheritance*. It allows a (sub)class to inherit variables and methods from superclasses. Some languages, such as Java, only support single inheritance for classes.

Subclasses can *override* some methods, changing the behavior inherited from superclasses. We use C++ term *virtual functions* to refer to methods that can be overridden. Virtual functions are *dynamically dispatched*—the actual function to invoke is selected based on the dynamic types of parameters. Java only supports single dynamic dispatch, i.e., the function is selected based only on the type of the *receiver* object.

Alloy [9] 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) [8]. 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.

Alloy supports some features of object-oriented design. However, Alloy does not have built in support for dynamic dispatch. Recently, Jackson and Fekete [7] presented an approach for modeling parts of Java in Alloy, pointing out that modeling "the notion of equality is problematic".

L.-H. Eriksson and P. Lindsay (Eds.): FME 2002, LNCS 2391, pp. 234–251, 2002.

In Java, the `equals` method, which allows comparing object values, as opposed to using the '`==`' operator, which compares object identities, is overridden in majority of classes. Good programming methodology suggests that `equals` be overridden in all immutable classes [14]. This method is pervasively used, for example in the Java Collections Framework [22] for comparing elements of collections. Any `equals` method must satisfy a set of properties, such as implementing an equivalence relation; otherwise, the collections do not behave as expected. However, getting `equals` methods right is surprisingly hard.

We present VAlloy, a veneer onto Alloy that enables intuitive modeling of dynamic dispatch. VAlloy introduces in Alloy virtual functions and related inheritance constructs. We give VAlloy a formal semantics through a translation to Alloy. The translation is similar to compilation of object-oriented languages, involving creation of virtual function tables. Since VAlloy models can be automatically translated to Alloy, they can also be automatically analyzed using the existing AA.

Having an easy way to model dynamic dispatch is important for several reasons. First, it enables automatic analysis of models of overridden methods. Second, it allows modeling comparisons based on object values and developing specifications for collections that use these comparisons, such as Java collections. Third, such specifications can be used to test the actual implementations, for example using the TestEra framework [16].

The rest of this paper is organized as follows. Section 2 gives an example that illustrates the key constructs of VAlloy. Section 3 defines a semantics for VAlloy through a translation to Alloy. Section 4 presents VAlloy specifications that partially model Java-like collections. Section 5 discusses some extensions to the key constructs of VAlloy. Section 6 reviews related work, and Section 7 presents our conclusions. The Appendix describes the basics of Alloy and the Alloy Analyzer.

2 Example

We illustrate VAlloy by modeling and analyzing an (in)correct overriding of the `equals` method in Java. We first develop an Alloy specification that contains only one `equals` method, and then describe challenges that arise in modeling method overriding. Finally, we present how VAlloy tackles these challenges.

2.1 Modeling `Equals` in Alloy

Consider the following `equals` method that appears in `java.awt.Dimension` in the standard Java libraries [22]:

```
class Dimension {
    int width;
    int height;
    public boolean equals(Object obj) {
        if (!(obj instanceof Dimension))
            return false;
        Dimension d = (Dimension)obj;
```

```
        return (width == d.width) && (height == d.height);
    }
}
```

We develop in Alloy (not yet VAlloy) a specification for the above method. An Alloy specification consists of a sequence of paragraphs that either introduce an *uninterpreted type* or express constraints over the types. We start with the following declarations:

```
sig Object {}                    // java.lang.Object
sig Dimension extends Object {   // java.awt.Dimension
  width: Integer,
  height: Integer
}
```

Each *signature*, introduced by the keyword `sig`, denotes a set of atomic individuals. In this specification, atoms in `sig Object` model Java objects. The signature `Dimension` is declared it to be a subset of `Object`. Alloy subsets model Java subclassing with typing rules being as follows.

Signatures declared without `extends` are *basic* signatures. Basic signatures are disjoint from one another and represent Alloy types. Subsets do not introduce new Alloy types. The type of an atom is the basic signature it belongs to; all atoms in the above specification, including those in `Dimension`, have Alloy type `Object`. We can reconstruct Java type, i.e., class, of modeled Java objects based on their (sub)set membership.

Fields `width` and `height` introduce relations between `Dimension` atoms and `Integer` atoms, where `Integer` is predefined in Alloy. More precisely, each field introduces a function that maps `Dimension` atoms to `Integer` atoms.

We next add to the specification a model of the above `equals` method:

```
fun Dimension::equals(obj: Object) {
  obj in Dimension               // instanceof
  this.width = obj.width && this.height = obj.height
}
```

The Alloy *function* `equals` records constraints that can be invoked elsewhere in the specification. This function has two arguments: `obj` and the implicit `this` argument, introduced with '`::`'. The function body constrains `obj` to be an atom of `Dimension`, effectively modeling Java's `instanceof`. This constraint is conjoined with the other that requires the fields to be the same. However, the above declaration does not constrain `this` to be an atom of `Dimension`; the declaration is equivalent to `fun Object::equals(obj: Object)`.

We next use the Alloy Analyzer (AA) to automatically check properties of the above specification. Each `equals` method should satisfy a set of properties: implement an equivalence relation and be consistent with `hashCode` [22]. The following Alloy assertion requires the function `equals`, which models the method `equals`, to be an equivalence relation:

```
assert equalsIsEquivalence {
  all o: Object |             // reflexivity
    o..equals(o)
  all o1, o2: Object |        // symmetry
    o1..equals(o2) => o2..equals(o1)
  all o1, o2, o3: Object |    // transitivity
    o1..equals(o2) && o2..equals(o3) => o1..equals(o3)
}
```

The operator '..' invokes Alloy functions (using static resolution). AA checks the above assertion and reports that there are no counterexamples.

2.2 Overriding

Consider `Dimension3D`, a subclass of `java.awt.Dimension` that adds a field `depth` and overrides `equals`:

```
class Dimension3D extends java.awt.Dimension {
    int depth;
    boolean equals(Object obj) {
        if (!(obj instanceof Dimension3D))
            return false;
        Dimension3D d = (Dimension3D)obj;
        return super.equals(obj) && depth = d.depth;
    }
}
```

In order to check the `equals` method in `Dimension3D`, we would like to add the following to the Alloy specification presented so far:

```
sig Dimension3D extends Dimension {
  depth: Integer
}
// duplicate function names are NOT allowed in Alloy
fun Dimension3D::equals(obj: Object) {
  obj in Dimension3D
  // super.equals needs to be inlined because
  // there is no built in support for super
  this.width = obj.width && this.height = obj.height
  this.depth = obj.depth
}
```

However, this does not produce the intended model of overriding. In fact, this is not even a legal Alloy specification—each Alloy specification must have unique function names.[1] We could try renaming one of the `equals` functions, but it does not directly solve the problem of modeling overriding. Namely, the invocations `o..equals(o')` should choose the function based on the Java type/class of `o`. Since Alloy has no built in support for dynamic dispatch, we would need to model it manually for each function. Instead, we propose that it be done automatically.

2.3 Modeling `Equals` in VAlloy

VAlloy introduces a natural way to model dynamic dispatch in Alloy. The following VAlloy specification models the above Java classes:

```
class Object {}
virtual fun Object::equals(obj: Object) { this = obj }

class Dimension {
  width: Integer,
  height: Integer
}
virtual fun Dimension::equals(obj: Object) {
  obj in Dimension
  this.width = obj.width && this.height = obj.height
}
```

[1] That is why we do not initially add `equals` function for `Object`.

```
class Dimension3D extends Dimension {
  depth: Integer
}
virtual fun Dimension3D::equals(obj: Object) {
  obj in Dimension3D
  super..equals(obj) && this.depth = obj.depth
}
```

The `class` declaration in VAlloy corresponds to the Alloy declaration `disj sig`, where `disj` indicates that the declared subset is disjoint from other `disj` subsets of its parent set. As in Java, VAlloy classes by default extend `Object`.

The `virtual` function modifier[2] is the main VAlloy extension to Alloy. This modifier declares a function that is dynamically dispatched at invocation, based on the VAlloy class of the receiver. VAlloy allows virtual functions to have the same name. The above example also shows the keyword `super` that VAlloy provides for modeling `super` as found in Java.

2.4 Checking VAlloy Specifications

Every VAlloy specification can be automatically translated into an Alloy specification. Section 3 presents the translation and the resulting Alloy specification for our running example.[3]

We use AA to automatically check the above assertion `equalsIsEquivalence`. Note that the invocations in the assertion do not need to change; the translation properly models dynamic dispatch. AA generates a counterexample[4]:

```
Object_2: Dimension3D {
  width = 0,
  height = 1,
  depth = 2
}
Object_1: Dimension {
  width = 0,
  height = 1
}
```

These two objects violate the symmetry property: `Object_1..equals(Object_2)`, but *not* `Object_2..equals(Object_1)`. This is because `equals` of `Dimension` is oblivious of the field `depth` declared in `Dimension3D`. This counterexample shows that it is hard to extend the `java.awt.Dimension` class and preserve the properties of `equals`.

A way to provide an overridable implementation of `equals` in Java is to use the `getClass` method instead of the `instanceof` primitive [18]. In the running example, it requires changing `equals` of `java.awt.Dimension` to use the expression `obj.getClass() == this.getClass()` instead of `obj instanceof Dimension`. A similar change should be made in `Dimension3D`, unless it is declared `final`, and therefore cannot be extended.

[2] VAlloy borrows the modifier name from C++.

[3] We have not yet implemented the translation; we perform it manually.

[4] AA took 5 seconds (including its boot-up time) using a scope of 3 atoms in each basic signature on a Pentium III 700 MHz with 256MB RAM.

Modeling this change in VAlloy is straightforward: change `obj in Dimension` with `obj..getClass() = this..getClass()` in the function `Dimension::equals`. VAlloy provides the function `getClass` that models the `final` method `getClass` from the class `java.lang.Object`. We translate the changed VAlloy specification into Alloy and again use AA to check the equivalence assertion. This time AA reports that there are no counterexamples.

3 VAlloy

This section presents VAlloy as an extension to Alloy. We define a formal semantics for VAlloy by giving a translation of VAlloy specifications to Alloy specifications. Details of Alloy semantics can be found in [9].

VAlloy adds the following to Alloy:

- `virtual` function modifier that declares a function whose invocation depends on the class of the receiver;
- `class` declaration that introduces VAlloy classes;
- `super` keyword that directly correspond to Java;
- `getClass` function that corresponds to the `getClass` method of the class `java.lang.Object`.

These constructs are syntactically added to Alloy in the obvious way.

3.1 Translation Example

We give a semantics to the new constructs through a translation into Alloy. The translation algorithm operates in six steps, which we first describe through examples. Figures 1 and 2 show the Java code and VAlloy specification from Section 2. For this example, the translation proceeds as follows.

Step 1. Compute the hierarchy of `class` declarations:

```
Object
 +-- Dimension
      +-- Dimension3D
```

Step 2. Construct `sig Class` and `sig Object` based on the above hierarchy:

```
sig Class { ext: option Class }
static part sig Object_Class, Dimension_Class,
                Dimension3D_Class extends Class {}
fact Hierarchy {
  no Object_Class.ext
    Dimension_Class.ext = Object_Class
      Dimension3D_Class.ext = Dimension_Class
}
sig Object { class: Class }
fact ObjectClasses {
  (Object - Dimension).class = Object_Class
    (Dimension - Dimension3D).class = Dimension_Class
      Dimension3D.class = Dimension3D_Class
}
fun Object::getClass(): Class { result = this.class }
```

```
class Object {
    boolean equals(obj: Object) {
        return this == obj;
    }
}
class Dimension {
    int width;
    int height;
    boolean equals(obj: Object) {
        if (obj.getClass() != this.getClass())
            return false;
        Dimension d = (Dimension)obj;
        return width == d.width &&
               height == d.height;
    }
}
class Dimension3D extends Dimension {
    int depth;
    boolean equals(obj: Object) {
        if (obj.getClass() != this.getClass())
            return false;
        Dimension3d d = (Dimension3d)obj;
        return super.equals(obj) &&
               depth == d.depth;
    }
}
```

Fig. 1. Java code

```
class Object {}
virtual fun Object::equals(obj: Object) {
  this = obj
}

class Dimension {
  width: Integer,
  height: Integer
}
virtual fun Dimension::equals(obj: Object) {
  obj..getClass() = this..getClass()
  this.width = obj.width
  this.height = obj.height
}
class Dimension3D extends Dimension {
  depth: Integer
}
virtual fun Dimension3D::equals(obj: Object) {
  obj..getClass() = this..getClass()
  super..equals(obj)
  this.depth = obj.depth
}

assert equalsIsEquivalence {
  all o: Object |              // reflexivity
    o..equals(o)
  all o1, o2: Object |         // symmetry
    o1..equals(o2) => o2..equals(o1)
  all o1, o2, o3: Object |     // transitivity
    o1..equals(o2) && o2..equals(o3) =>
    o1..equals(o3)
}
```

Fig. 2. VAlloy specification

```
sig Class { ext: option Class }
static part sig
  Object_Class, Dimension_Class,
  Dimension3D_Class extends Class {}
fact Hierarchy {
  no Object_Class.ext
    Dimension_Class.ext = Object_Class
      Dimension3D_Class.ext = Dimension_Class
}
sig Object { class: Class }
fact ObjectClasses {
  (Object - Dimension).class = Object_Class
    (Dimension - Dimension3D).class =
       Dimension_Class
      Dimension3D.class = Dimension3D_Class
}
fun Object::getClass(): Class {
  result = this.class
}
fun Object::equals(obj: Object) {
  this.class = Object_Class =>
  this..Object_equals(obj)
    this.class = Dimension_Class =>
    this..Dimension_equals(obj)
      this.class = Dimension3D_Class =>
      this..Dimension3D_equals(obj)
}

fun Object::Object_equals(obj: Object) {
  this = obj
}

disj sig Dimension extends Object {
  width: Integer,
  height: Integer
}
fun Object::Dimension_equals(obj: Object) {
  obj..getClass() = this..getClass()
  this.width = obj.width
  this.height = obj.height
}

disj sig Dimension3D extends Dimension {
  depth: Integer
}
fun Object::Dimension3D_equals(obj: Object) {
  obj..getClass() = this..getClass()
  this..Dimension_equals(obj)
  this.depth = obj.depth
}

assert equalsIsEquivalence {
  all o: Object |              // reflexivity
    o..equals(o)
  all o1, o2: Object |         // symmetry
    o1..equals(o2) => o2..equals(o1)
  all o1, o2, o3: Object |     // transitivity
    o1..equals(o2) && o2..equals(o3) =>
    o1..equals(o3)
}
```

Fig. 3. Translated Alloy specification

Atoms in `Class` and the *fact* `Hierarchy` represent the VAlloy `class` declarations. (A `fact` in Alloy expresses constraints that must hold for all instances of the specification.) For each atom `c` in `Class`, `c.ext` gives the `Class` atom that corresponds to the superclass of `c`.[5] The keyword `static` constrains each of the declared subsets to contain exactly one atom, and the keyword `part` declares a partition—the subsets are disjoint and their union is the whole set.

For each atom `o` in `Object`, `o.class` gives the corresponding `Class` atom. This correspondence is set with `fact ObjectClasses` based on the VAlloy `class` hierarchy. (The '`-`' operator denotes set difference in Alloy.) This translation step also introduces the function `getClass`.

Step 3. Change `class` declarations into `disj sig` declarations, adding `extends Object` where required:

```
disj sig Dimension extends Object { ... }
disj sig Dimension3D extends Dimension { ... }
```

This step does not change field declarations.[6]

Step 4. Rename each virtual function so that all functions in the specification have unique names:

```
fun Object::Object_equals(obj: Object) { this = obj }
fun Object::Dimension_equals(obj: Object) { ... }
fun Object::Dimension3D_equals(obj: Object) { ... }
```

This step also removes the modifier `virtual`, translating dynamically dispatched VAlloy functions into statically dispatched Alloy functions.

Step 5. Add, for each overridden function name, a *dispatching* function, i.e., a new Alloy function that models dynamic dispatch:

```
fun Object::equals(obj: Object) {
  this.class = Object_Class =>
  this..Object_equals(obj)
    this.class = Dimension_Class =>
    this..Dimension_equals(obj)
      this.class = Dimension3D_Class =>
      this..Dimension3D_equals(obj)
}
```

This step is the crux of the translation. It allows function invocations in VAlloy to be written in the usual Alloy notation, but it models dynamic dispatch semantics—the actual function is selected based on the class of the receiver.

Step 6. Replace each invocation on `super` with an invocation to the corresponding, previously renamed, static function:

```
fun Object::Dimension3D_equals(obj: Object) {
  obj..getClass() = this..getClass()
  this..Dimension_equals(obj) && this.depth = obj.depth
}
```

[5] For simplicity, we only present single inheritance, where the hierarchy can only be a tree. In multiple inheritance, each class can have a set of superclasses.

[6] For simplicity, we do not present modeling `null`, which would require slightly changing field declarations.

This completes the translation. Figure 3 shows the full resulting Alloy specification. Note that the translation does not change the assertion; the invocations `o..equals(o')` remain written in the most intuitive manner, but they have dynamic dispatch semantics.

3.2 General `Class` Hierarchy

To illustrate the general translation of `class` hierarchy, consider the following excerpt from a VAlloy specification:

```
class O {}                 virtual fun O::hC() { /*O*/ }
  class C extends O {}     virtual fun C::hC() { /*C*/ }
    class C1 extends C {} // C1 does not define fun hC
    class C2 extends C {} virtual fun C2::hC() { /*C2*/ }
  class D extends O {}     // D does not define fun hC
    class D1 extends D {} virtual fun D1::hC() { /*D1*/ }
```

For this hierarchy, the translation generates the following `sig Class` and `sig Object`:

```
sig Class { ext: option Class }
static part sig O_Class, C_Class, C1_Class, C2_Class,
                D_Class, D1_Class extends Class {}
fact Hierarchy {
  no O_Class.ext
    C_Class.ext = O_Class
      C1_Class.ext = C_Class
      C2_Class.ext = C_Class
    D_Class.ext = O_Class
      D1_Class.ext = D_Class
}
sig Object { class: Class }
fact ObjectClasses {
  (O - C - D).class = O_Class
    (C - C1 - C2).class = C_Class
      C1.class = C1_Class
      C2.class = C2_Class
    (D - D1).class = D_Class
      D1.class = D1_Class
}
```

For the function `hC`, the translation generates the following Alloy functions:

```
fun O::O_hC() { /*O*/ }
fun O::C_hC() { /*C*/ }
// there is no O::C1_hC()
fun O::C2_hC() { /*C2*/ }
// there is no O::D_hC()
fun O::D1_hC() { /*D1*/ }
fun O::hC() {
  this.class = O_Class => this..O_hC()
    this.class = C_Class => this..C_hC()
      this.class = C1_Class => this..C_hC() /* not C1 */
      this.class = C2_Class => this..C2_hC()
    this.class = D_Class => this..O_hC() /* not D */
      this.class = D1_Class => this..D1_hC()
}
```

3.3 Summary

To summarize, the translation from VAlloy to Alloy proceeds in the following six steps:

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

4 Collections

This section presents VAlloy models for some collection classes. Our main focus is comparison based on object values. We ignore the orthogonal issue of modeling state, i.e., sharing and object interactions. An approach for modeling state in Alloy is discussed in [7], and we can apply the same approach to VAlloy.

We first present a specification for sets and then reuse it to specify maps. Finally, using a tree-based implementation of sets, we show how properties of abstract data types can be expressed in VAlloy.

4.1 Sets

We develop a VAlloy specification for sets whose membership is based on object values, not object identities. As in Java, elements of the sets are objects of classes that (in)directly extend `Object` and potentially override `equals`.

We first declare a VAlloy class for sets:

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

For each atom `a` in `Set`, `a.s` is the (Alloy) set of objects in the (modeled) set `a`. To constrain set membership to be based on object values, we introduce the following `fact`:

```
fact SetIsBasedOnEquals {
  all a: Set | all disj e1, e2: a.s | !e1..equals(e2)
}
```

This `fact` requires distinct elements in each set to be not equal with respect to `equals`. For example, this rules out the set `a` such that `a.s={d1,d2}`, where `d1` and `d2` are distinct atoms (i.e., `d1!=d2`) of `Dimension`, but `d1.width=3`, `d1.height=8` and also `d2.width=3`, `d2.height=8`, which makes `d1..equals(d2)`. Note that `a.s` is a valid Alloy set.

It is now easy to specify some set functions from the `java.util.Set` interface:

```
virtual fun Set::contains(o: Object) {
  some e: this.s | o..equals(e)
}
virtual fun Set::add(o: Object): Set {
  this.s..contains(o) =>
    result.s = this.s,
    result.s = this.s + o
}
virtual fun Set::remove(o: Object): Set {
  result.s = this.s - { e: this.s | e..equals(o) }
}
virtual fun Set::isEmpty() { no this.s }
```

```
virtual fun Set::clear(): Set { no result.s }
virtual fun Set::size(): Integer { result = #this.s }
virtual fun Set::subset(a: Set) {
  all e: this.s | a..contains(e)
}
virtual fun Set::equals(o: Object) {
  o in Set
  o..size() = this..size()
  o..subset(this)
}
```

The most interesting function is `equals`, which compares two sets for equality. It checks that both sets have the same number of elements and that `o` is a subset (based on `equals`) of `this`. The function `remove` uses set comprehension to specify an object's removal from a set.

The above VAlloy specification closely models `java.util.Set`. The main difference is that this specification is written in a functional style and does not model state modifications. As mentioned, state can be modeled using the approach from [7], which also presents a way to handle iterators. Therefore, we do not model "bulk operations" on sets, such as `addAll`, based on iterators. Instead, we present an analogous function for set union:

```
virtual fun Set::union(a: Set): Set {
  this..subset(result)
  a..subset(result)
  all e: result.s | this..contains(e) || a..contains(e)
}
```

Note that the use of `contains` (and `subset` based on `contains`), which is based on `equals`, enables specifying `union` in a direct way.

4.2 Maps

We next develop a partial VAlloy specification for maps, such as `java.util.Map`, that compare keys based on `equals`. In this specification, we reuse the `class Set` defined above to automatically constrain the set of keys:

```
class Map {
  keys: Set
  map: keys.s ->! Object
}
```

We model the mapping from keys to values using an Alloy relation `map`; the *multiplicity marking* '`!`' indicates that for each key, there is exactly one `Object`. For each atom `a` in `class Map`, `a.map` is the actual mapping. For a key `k`, `a.map[k]` gives the value that `k` is mapped to in map `a`.

We next model the essential map functions:

```
virtual fun Map::get(key: Object): Object {
  this.keys..contains(key) => result = this.map[key]
}
virtual fun Map::put(key: Object, value: Object): Map {
  result.keys = (this.key - { e: this.keys | e..equals(key) }) + key,
  result.map = (this.map - { e: this.keys | e..equals(key) }->Object) + key->value
}
```

The function `get` returns the value that `key` is mapped to, if such a key exists in the map; otherwise, the behavior of `get` is 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.

4.3 Trees

We next use a tree-based implementation of sets to illustrate how properties of abstract data types can be expressed in VAlloy. Consider the following declaration for binary trees:

```
class Tree { root: Node }
class Node {
  left: Node,
  right: Node,
  data: Object
}
```

Suppose that these VAlloy trees model a Java implementation of sets based on `equals`. We can state the *abstraction function* [14] for these trees in VAlloy:

```
fun Tree::abstractionFunction(): Set {
  result.s = this.root.*(left+right).data
}
```

The '`*`' operator is reflexive transitive closure, and `root.*(left+right)` denotes an (Alloy) set of all `Node`s reachable from the `root`. The set of `Object`s from those nodes is obtained accessing `data`, and the abstraction function constrains this Alloy set to be a `Set`.

We also use VAlloy to state *representation invariant* [14] for these trees. Assume that they have the following structural constraints: root nodes are sentinels (and thus never `null`) and leaf nodes point to themselves. The following `repOk` predicate characterizes the representation invariants for a tree:

```
fun Tree::repOk() {
  // no node points to root
  no this.root.~(left+right)
  // acyclic (with self loops for leafs)
  all n: this.root.*(left+right) {
    n.left = n || n !in n.left.*(left+right)
    n.right = n || n !in n.right.*(left+right)
  }
  // no duplicates w.r.t equals()
  some a: Set | a = this..abstractionFunction()
}
```

(The '~' operator denotes transpose of a binary Alloy relation.) Beside the structural invariants, a valid tree is required to be a concrete representation of some `Set`. Note how the `equals` constraints from the abstract representation, `Set`, propagate to the concrete representation, `Tree`.

5 Extensions

VAlloy presents our first step toward modeling in Alloy advanced constructs from object-oriented languages. The main focus has been on method overriding in Java. We have therefore designed VAlloy to support subclasses that can arbitrarily change behavior of inherited methods.

Our approach can easily be extended to support intuitive modeling of multiple inheritance, such as in C++, and multi-method dispatch, such as in Cecil. Support for method overloading can clearly be added through a simple syntactic manipulation. We omitted support for Java's interfaces, keeping in line with

Alloy's "micromodularity" philosophy of being a lightweight language amenable to fully automatic analysis. Similarly, we do not consider encapsulation.

We have recently developed some recipes to model in Alloy several other common imperative programming constructs like mutation, statement sequencing, object allocation, local variables, and recursion. We have used these recipes to design AAL [12], an annotation language based on Alloy for Java programs. AAL offers both fully automatic compile-time analysis using the Alloy Analyzer and dynamic analysis through generation of run-time assertions.

We would like to develop further recipes, for example, for modeling in Alloy the exceptional behavior of methods. Having exceptions would also allow modeling arrays with bound checking. We are also considering adding support for modeling multi-threading.

To explore practical value of VAlloy, we intend to implement the translation and use VAlloy in connection with some existing frameworks. Daikon [3] is a tool for dynamically detecting likely program invariants; we are considering to use it to detect (partial) VAlloy specifications of Java classes. TestEra [16] is a framework for automated test generation and correctness evaluation of Java classes; we are considering to use VAlloy specifications for TestEra.

6 Related Work

Recently, Jackson and Fekete [7] 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. Their approach is orthogonal to our handling of virtual functions; we are planning to combine these two approaches.

Alloy has been used to check properties of programs that manipulate dynamic data structures. Jackson and Vaziri [10] 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 [16], 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, which use comparisons based on object values, requires modeling the `equals` method in Alloy. This led us to tackle modeling general Java-like inheritance in Alloy. VAlloy presents some ideas toward that goal.

The Java Modeling Language (JML) [13] 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 [23] 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 [20] extends the Z specification language [21], 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. [11] add inheritance to the IOA language [15] 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.

7 Conclusions

We described VAlloy, a veneer onto the first order, relational language Alloy. All function invocations in Alloy are static; Alloy has no direct support for dynamic dispatch. VAlloy introduces virtual functions in Alloy, which enables intuitive modeling of inheritance, such as that of Java. We illustrated the use of VAlloy by modeling a small part of the Java Collections Framework.

We defined a formal semantics for VAlloy through a translation to Alloy. VAlloy models can be automatically translated into Alloy. The translation is similar to building virtual function tables for object-oriented languages and can benefit from optimizations based on class hierarchy. The translated specifications can be automatically checked using the existing Alloy Analyzer. We believe that VAlloy can be effectively used for specification and checking of Java classes.

Acknowledgments. We would like to thank Manu Sridharan and Ang-Chih Kao for comments on an earlier draft of this paper. This work was funded in part by ITR grant #0086154 from the National Science Foundation.

References

1. J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.
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. Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.
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.
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. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 2002. (to appear).
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Seattle, WA, Nov 2002.
13. 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).
14. Barbara Liskov. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2000.
15. Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
16. 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.
17. Chris Moss. *Prolog++ The Power of Object-Oriented and Logic Programming.* Addison-Wesley, 1994.

18. Mark Roulo. How to avoid traps and correctly override methods from `java.lang.Object`. http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html.
19. Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
20. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
21. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
22. Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. http://java.sun.com/j2se/1.3/docs/api/.
23. 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.

A Alloy

In this section we describe the basics of the Alloy specification language and the Alloy Analyzer; details can be found in [5,6,8]. Alloy is a strongly typed language that assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. An Alloy model is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new types, and a variety of formula paragraphs, used to record constraints.

A.1 Signature Paragraphs

A signature paragraph introduces a basic type and a collection of relations (that are called *fields*) in it along with the types of the fields and constraints on their values. For example,

```
sig Class{
  ext: option Class
}
```

introduces `Class` as an uninterpreted type (or a set of atoms). The field declaration for `ext` introduces a relation from `Class` to `Class`. This relation is a partial function as indicated by the keyword `option`: for each atom `c` of `Class`, `c.ext` is either an atom of `Class` or the empty set. In a field declartion, the keyword `set` can be used to declare an arbitrary relation; ommiting a keyword declares a total function.

A signature may inherit fields and constraints from another signature. For example,

```
static part sig Object_Class, Dimension_Class, Dimension3D_Class extends Class {}
```

declares `Object_Class`, `Dimension_Class`, and `Dimension3D_Class` to be subsets of `Class` and inherit the field `ext`. The keyword `part` declares these subsets to be disjoint and their union to be `Class`; `disj` declares disjoint subsets. In a signature declartion, the keyword `static` specifies the declared signature(s) to (each) contain exactly one element.

A.2 Formula Paragraphs

Formula paragraphs are formed from Alloy expressions.

Relational expressions. The value of any expression in Alloy is always a relation—that is a collection of tuples of atoms. Each element of such a tuple is atomic and belongs to some basic type. A relation may have any arity greater than one. Relations are typed. Sets can be viewed as unary relations.

Relations can be combined with a variety of operators to form expressions. The standard set operators—union (`+`), intersection (`&`), and difference (`-`)—combine two relations of the same type, viewed as sets of tuples. The dot operator is relational composition. When `p` is a unary relation (i.e., a set) and `q` is a binary relation, `p.q` is standard composition; `p.q` can alternatively be written as `q[p]`, but with lower precedence. The unary operators `~` (transpose), `^` (transitive closure), and `*` (reflexive transitive closure) have their standard interpretation and can only be applied to binary relations.

Formulas and declarations. Expression quantifiers turn an expression into a formula. The formula `no e` is true when `e` denotes a relation containing no tuples. Similarly, `some e`, `sole e`, and `one e` are true when `e` has some, at most one, and exactly one tuple respectively. Formulas can also be made with relational comparison operators: subset (written `:` or `in`), equality (`=`) and their negations (`!:`, `!in`, `!=`). So `e1:e2` is true when every tuple in (the relation denoted by the expression) `e1` is also a tuple of `e2`. Alloy provides the standard logical operators: `&&` (conjunction), `||` (disjunction), `=>` (implication), and `!` (negation); a sequence of formulas within curly braces is implicitly conjoined.

A *declaration* is a formula `v op e` consisting of a variable `v`, a comparison operator `op`, and an arbitrary expression `e`. Quantified formulas consist of a quantifier, a comma separated list of declarations, and a formula. In addition to the universal and existential quantifiers `all` and `some`, there is `sole` (at most one) and `one` (exactly one). In a declaration, `part` specifies partition and `disj` specifies disjointness; they have their usual meaning.

A set marking is one of the keywords `scalar`, `set` or `option`, prefixing the expression. The keyword `scalar` adds the side condition that the variable denotes a relation containing a single tuple; `set` says it may contain any number of tuples; `option` says it contains at most one tuple. The default marking is `set`, except when the comparison operator is the colon(`:`) or negated colon (`!:`), and the expression on the right is unary, in which case it is `scalar`.

A relation marking is one of the symbols `!`, `?`, and `+` read *exactly one*, *at most one*, and *one or more* respectively. These markings are applied to the left and right of an arrow operator. Suppose a relation `r` is declared as

```
r : e1 m -> n e2
```

where `m` and `n` are relation markings. The markings are interpreted as imposing a side condition on `r` saying that for each tuple t_1 in `e1`, there are `n` tuples t_2 in e2 such that t_1t_2 appears in r, and for each tuple t_2 in e2, there are m tuples t_1 such that t_1t_2 appears in r.

The declaration

```
disj v1,v2,... : e
```

is equivalent to a declaration for each of the variables v1,v2,..., with an additional constraint that the relations denoted by the variables are disjoint (i.e., share no tuple); the declaration part additionally makes their union e.

A.3 Functions, Facts, and Assertions

A function (fun) is a parametrized formula that can be "applied" elsewhere. A fact is a formula that takes no arguments and need not be invoked explicitly; it is always true. An assertion (assert) is a formula whose correctness needs to be checked, assuming the facts in the model.

A.4 Alloy Analyzer

The Alloy Analyzer [8] (AA) is an automatic tool for analyzing models created in Alloy. Given a formula and a *scope*—a bound on the number of atoms in the universe—AA determines whether there exists a model of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it. Since first order logic is undecidable, AA limits its analysis to a finite scope.

AA's analysis [8] is based on a translation to a boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers.

AA provides two kinds of analysis: *simulation* in which the consistency of a fact or function is demonstrated by generating a snapshot showing its invocation, and *checking*, in which a consequence of the specification is tested by attempting to generate a counterexample.

AA can enumerate all possible instances of an Alloy model. AA adapts the symmetry-breaking predicates of Crawford et al. [1] to provide the functionality of reducing the total number of instances generated—the original boolean formula is conjugated with additional clauses in order to produce only a few instances from each isomorphism class [19].