An Analyzable Annotation Language

Sarfraz Khurshid Darko Marinov Daniel Jackson

Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139
{khurshid,marinov,dnj}@lcs.mit.edu

Abstract

The Alloy Annotation Language (AAL) is a language (under development) for annotating Java code based on the Alloy modeling language. It offers a syntax similar to the Java Modeling Language (JML), and the same opportunities for generation of run-time assertions. In addition, however, AAL offers the possibility of fully automatic compile-time analysis. Several kinds of analysis are supported, including: checking the code of a method against its specification; checking that the specification of a method in a subclass is compatible with the specification in the superclass; and checking properties relating method calls on different objects, such as that the equals methods of a class (and its overridings) induce an equivalence. Using partial models in place of code, it is also possible to analyze object-oriented designs in the abstract: investigating, for example, a view relationship amongst objects.

The paper gives examples of annotations and such analyses. It presents (informally) a systematic translation of annotations into Alloy, a simple first-order logic with relational operators. By doing so, it makes Alloy's automatic analysis, which is based on state-of-the-art SAT solvers, applicable to the analysis of object-oriented programs, and demonstrates the power of a simple logic as the basis for an annotation language.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.1 [**Software Engineering**]: Specifications; D.2.4 [**Software Engineering**]: Program Verification—Class invariants, programming by contract, formal methods.

General Terms

Verification, Languages

Keywords

Specification language, Java language, Alloy modeling language, Alloy Analyzer, compile-time analysis

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

1 Introduction

The benefits of annotating code with specifications are well known, but for the most part they have not been realized, and programmers still view them as more trouble than they are worth. Much progress has been made. Annotation languages are now better integrated with programming languages; they can handle the complexities of object-oriented code (such as dynamic dispatch, and the interaction between frame conditions and subclassing); and they can be readily compiled into run-time assertions. But to make annotations attractive to practitioners, we believe it is necessary to squeeze more value from them, by providing new analyses for the same annotations.

In this paper, we outline an annotation language, called AAL, that we are currently developing. The paper illustrates AAL on small examples. We have not yet demonstrated that the features of the language, or its analysis technology, will scale to realistic programs. AAL is similar to the Java Modeling Language (JML) [19] in its aim of providing a lightweight approach to code annotation, and it has adopted the JML syntax for distinguishing pre- and post-state values. JML, however, uses Java expression syntax, and is conceptually founded on algebraic specifications [18]; AAL, in contrast, is based on a simple first-order logic with relational operators, and is thus more in the tradition of semantic data modeling (now called "object modeling").

AAL allows partial specifications of methods to be written, which can be executed as run-time assertions. In addition, however, it supports fully automatic compile-time analysis. This analysis, which is based on our previous work, and reduces to finding models of relational formulas, has a variety of applications. It can be used to check code against specifications, producing counterexample traces that show how a method's code misbehaves. It can be used to generate test cases from invariants and preconditions fully automatically; this is especially useful for elaborate data structures, such as trees, which cannot be generated randomly because of intricate structural constraints, and are tedious to generate manually. It can be used also for more elaborate checks that are easily expressed in our logic but not usually expressible at all in other annotation languages, such as that an equals method actually defines an equivalence.

Automatic compile-time analysis, more than run-time analysis, has the potential to change how programmers work. The success of test-first programming, a key practice of Extreme Programming [1], indicates not only the importance of hav-

ing regressions tests (and having them early), but also the value of focusing on intent before coding. Even though the programmer isn't required to write a specification, the act of devising test cases forces a consideration of what behavior is intended. Our approach promises to bring the benefit of test-first programming with lower cost. An invariant (especially for a complex structure) is usually much less work to write than a reasonable test suite. Having written it, the programmer can rely on our analysis not only to generate a high quality test suite but also to check crucial properties of the specification itself. A crucial advantage of compile-time analysis is that it can be applied before the code is complete, whereas run-time analyses require not only that the module be complete, but that whatever libraries it uses be complete and available also.

The language we present is a declarative, first-order logic. This makes it simple and familiar. The inclusion of relational operators makes it easy to express navigations amongst objects. The set of elements of a list, for example, may be denoted by the expression `p.*next.elt`, where `p` is the variable that points to the list, `*next` indicates zero or more traversals of a field `next` from list nodes to list nodes, and `elt` indicates traversal of a field from a node to its contained element. The user can view the heap as a labeled graph, modeled with standard notions of sets and relations. We believe this is simpler than an algebraic viewpoint, which requires a collection of predefined datatypes to model sets, tuples, sequences, and so on.

The logic is expressive enough to capture much of the behavior of an object-oriented program. In particular, it can express the mutation of shared objects that is the cause of much complexity in object-oriented programs. The annotation language is actually a full modeling language, and can be used to model and analyze an object-oriented program in its entirety. We illustrate this by application to the design of *views*, in which mutation of one object is propagated to another belonging to a different class [7].

Previous work has shown how to use the Alloy language to represent the heap of an object-oriented program [12, 13], how to check code automatically against an Alloy specification [15], and how to generate test cases automatically from Alloy invariants [24]. The new contributions of this paper are:

- An embedding of the Alloy logic in an annotation language designed to be as close as possible to JML;
- A translation scheme that translates annotations and checks into simple Alloy formulas, making existing Alloy analysis technology [9] applicable;
- As part of this scheme, a handling of some of the tricky aspects of object-oriented programs and specifications, most notably: frame conditions (and their interaction with object sharing and subclassing), dynamic dispatch, recursion, and user-defined equality;
- A series of small examples illustrating the power of the approach, including analyzing whether a user-defined equality method obeys the object contract, and analyzing views (as used extensively in the Java Collections Framework [34]).

Our approach can be viewed as an attempt to gain the benefits of two complementary approaches to annotation: the JML approach [19], which allows rich specifications but only run-time analysis, and the Extended Static Checker (ESC) approach [4], which provides compile-time analysis but for much less expressive specifications. This work builds on both of these projects. From JML, it takes ideas of syntax and specification inheritance. From ESC, it takes ideas about how to interpret frame conditions. It combines these ideas with an analysis technology that makes it possible to have both rich specifications and compile-time analysis.

2 Overview

The key idea in this paper is that the behavior of an object-oriented program can be modeled and analyzed within a simple first-order logic of sets and relations. In this section, we explain briefly how the state of an object-oriented program is modeled with sets and relations; how a method's behavior is modeled as a formula; and how the problem of analysis always reduces to finding an assignment of (set and relation) values to variables that makes a formula true. In later sections, these ideas are elaborated more fully.

2.1 Modeling state

The heap of an executing program is viewed as a labeled graph whose nodes represent objects and whose edges represent fields. The presence of an edge labeled *f* from node *o* to *o'* says that the *f* field of the object *o* points to the object *o'*. Mathematically, we treat this graph as a set (the set of nodes) and a collection of relations, one for each field. From a type perspective, the relations are untyped, and so any object may be associated by a given field with any other. We therefore classify the objects into classes by associating a subset with each class, and we constrain the relation associated with a field so that it only maps objects that belong to the field's class.

This simple approach, in which fields are essentially untyped, has a great advantage. It eliminates the need for casts. The assertion that an object belongs to a given class becomes a membership test. And since field "get" (dereferencing) is treated as relational image, application of a field to an object of the wrong class yields the empty set. This is particularly handy in navigation expressions, in which a field is applied over a set of objects, only some of which belong to the appropriate class.

To model mutation, we simply associate a distinct graph with each state. In a specification there are only two states—the pre-state and the post-state. Mathematically, we treat fields as ternary relations, each of which maps an object to an object in a given state.

2.2 Specifications and models

Our annotation language includes the following parts. Each method may be annotated with three formulas—a precondition (labeled `requires`), a postcondition (labeled `ensures`),

and a behavior model (labeled `does`)—and a frame condition (labeled `modifies`). The precondition, postcondition and frame condition together form a *specification*, which is used to check the code of the method. In the checking of a client, the specification may be used as a surrogate. Sometimes, however, a client may legitimately rely on stronger properties than the specification guarantees, because, for example, the specification has been deliberately weakened to ensure that subclasses are subtypes. Or we may want to perform an analysis of the method that accounts for the details of its behavior. In these cases, it is more appropriate to use the behavior model than the specification. This model may be provided explicitly by the user (especially if no code is yet present), but more commonly it will have been automatically generated from the code, or from the specification (by conjoining pre- and post-conditions). We thus use the terms specification and (behavior) model interchangeably, except when we want to point out the distinction.

2.3 Dynamic dispatch

The specification or behavior model of a method may "invoke" other methods as if they were mathematical functions. This is necessary for specifying polymorphic methods. A method to insert an element into a set, for example, will use the `equals` method of the element to determine whether or not the element should be inserted. To express this in the specification or model, we use `equals` as if it were a mathematical function. In order to capture the dynamic dispatch in which the meaning of this function depends on the class the element belongs to, this function is desugared to a wrapper function that invokes the `equals` function of the appropriate class based on which class the element belongs to.

2.4 Analysis

Our analysis works by translating theorems to be checked into Alloy [10, 14], a software modeling language. We use Alloy's analysis tool, the Alloy Analyzer, to check the theorems. The analysis always reduces to finding an assignment for a formula that binds its free variables to values in such a way that the formula is true. In general, the free variables will represent the state of the program at several points in its execution. To check that a method in a subclass correctly preserves the specification of its superclass method, for example, we would construct a formula saying that a pre- and post-state exist that satisfy the constraints of the behavioral model of the subclass method, but do not satisfy the constraints of the superclass specification. A solution to this formula is a pair of states illustrating a violation: that is, a counterexample.

Our language is undecidable, so the analysis cannot be complete. The Alloy Analyzer conducts a search in a "scope" specified by the user that bounds the size of the basic types. In application to program checking, this amounts to limiting the size of the heap. A failure to find a counterexample does not mean that the theorem holds; it may mean that a counterexample exists, but only for heaps larger than those considered. In our experience in analyzing design models, we have found that surprisingly small scopes suffice to detect many subtle errors, and it seems likely that the same should hold for code.

The reader may wonder how this approach differs from testing; after all, only some finite set of cases is considered. There are two crucial differences. First, the user does not have to provide any test cases. Second, experience with model checking in the last decade has demonstrated that exhaustive analysis of a bounded space is far more effective at finding bugs than testing, largely because of the large number of cases considered. The satisfiability solvers that we use as the underlying technology in the Alloy Analyzer can routinely handle spaces of several hundred bits in width; an analysis of a billion cases (or, a state space of 30 bits), which is completely infeasible in testing, is thus very small.

3 Examples

Our presentation is organized around two examples. Both illustrate how programs that are seemingly straightforward can become surprisingly complicated. In the first, we consider determining whether the `equals` method of a class satisfies the "object contract"—in particular that it define an equivalence relation over the elements of the class. In practice, it is easy to get this wrong, especially when overriding the method in a subclass, because of the asymmetry of method dispatch.

In the second, we consider two implementations of a set interface: one a standard implementation as a red-black tree, and the other as a view on the first. This second class exhibits strange properties. Modifying the subset class causes modifications in the other; worse, because a subset is defined with a predicate on elements, inserting an element that does not satisfy the predicate will not cause it to be inserted at all. A set object thus fails to obey the simplest set axioms: that inserting an element into a set results in it belonging to the set afterwards. In fact, even the complexities of views are not necessary to break these axioms; since an inserted object may replace an object already present that is equivalent (by its `equals` method), insertion may actually remove an object! These are not contrived examples. It is for reasons such as these that the specification of the `Set` interface in the Java Collections Framework does not claim that the standard set axioms hold.

3.1 Overriding equals

We first use AAL to specify the behavior of two Java `equals` methods and to check the correctness of their overriding. Any implementation of `equals` should satisfy two crucial properties: at least being an equivalence relation and being consistent with `hashCode` [34]. It is easy to develop such implementations incorrectly [30], and special care should be taken for classes that allow `equals` to be overridden, i.e., for classes that are not `final`.

3.1.1 Superclass

Consider the `equals` method of the class `java.awt.Dimension` that is a non-`final` class from the standard Java libraries [34]. An object of `java.awt.Dimension` has two integer fields, `width` and `height`. The following is an example AAL annotation of `java.awt.Dimension` and its `equals` method:

```
class Dimension {
    int width;
    int height;

    //@ does {
    //@   \result = (obj instanceof Dimension &&
    //@               this.width = obj.width &&
    //@               this.height = obj.height)
    //@ }
    public boolean equals(Object obj) {
        if (!(obj instanceof Dimension))
            return false;
        Dimension d = (Dimension)obj;
        return (width == d.width) &&
               (height == d.height);
    }
}
```

The `does` annotation specifies a model for the behavior of the method. Such models can be provided by the programmer or automatically generated from the code, as here. A model is a formula over the method parameters, classes, and fields. The keyword `\result`, borrowed from JML, allows referring to the return value of the method. In this example, we consider an annotation that provides a model and not a specification, because we want to check that the model correctly implements the specification.

3.1.2 Checking AAL models

Every AAL specification and model can be automatically translated into Alloy and checked with the Alloy Analyzer (AA).[1] We describe in this section only the result of the analysis; Section 5 presents the translation. We use AA to automatically check that all `equals` methods implement an equivalence relation. The following AAL assertion expresses the equivalence property:

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

This assertion consists of three, implicitly conjoined, formulas. The variables quantified with `allnn` range over all non-`null` objects. The operator '`=>`' denotes implication. The operator '`..`' invokes AAL methods, using (single) dynamic dispatch as in Java, based on the dynamic class of the receiver object. Note that this property is inherently *global*, i.e., an invariant on all classes and *not* just a class invariant for `Object`.

The assertion is checked by presenting a translation into basic Alloy (described later in Section 5) to the Alloy Analyzer. The expressions involving `equals` are expanded using the behavior model of the `Dimension` class given above. In this case, the analyzer checks the assertion, and reports that there are no counterexamples for this particular class.

[1]We have not yet implemented the translation; for the examples of this paper we performed it manually.

3.1.3 Subclassing with overriding

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

```
class Dimension3D extends java.awt.Dimension {
    int depth;

    //@ does {
    //@   \result = (obj instanceof Dimension3D &&
    //@               super..equals(obj) &&
    //@               this.depth = obj.depth)
    //@ }
    boolean equals(Object obj) {
        if (!(obj instanceof Dimension3D))
            return false;
        Dimension3D d = (Dimension3D)obj;
        return super.equals(obj) &&
               depth = d.depth;
    }
}
```

We again use AA to check the assertion `equalsIsEquivalence`, and this time it automatically generates a counterexample:

```
Object_1: Dimension {
  width = 0,
  height = 1 }

Object_2: Dimension3D {
  width = 0,
  height = 1,
  depth = 3 }
```

These two objects violate the symmetry property: setting `o = Object_1` and `o' = Object_2` makes `o..equals(o')`, but *not* `o'..equals(o)`. The reason is that the former `equals` is from `Dimension`, and it is oblivious of the field `depth` introduced in `Dimension3D`, whereas the latter `equals` is from `Dimension3D`, and it correctly compares all the fields in the objects. This counterexample shows that `Dimension3D` incorrectly overrides `equals`, as it violates the equivalence property.

The problem is actually in the `equals` method of the class `java.awt.Dimension`; it is hard to correctly override it in a subclass. An overridable `equals` can be implemented in Java using the `getClass` method instead of the `instanceof` primitive [30]. In our running example, it requires changing `equals` of `java.awt.Dimension` to use the expression `obj.getClass() == this.getClass()` instead of `obj instanceof Dimension`.

Modeling this change in AAL is straightforward; we replace

```
obj instanceof Dimension
```

with

```
obj..getClass() = getClass()
```

in `does`. We translate the changed AAL specification into Alloy and again use AA to check the equivalence assertion. This time AA reports no counterexamples.

3.2 Subset views

We next illustrate AAL by developing specifications for sets and subset views. We consider a generalization of views returned by the `subSet` method in the class `java.util.TreeSet` of the Java Collections Framework (JCF). We develop example specifications for parts of the following:

- The `java.util.Set` interface.
- The abstract class `java.util.AbstractSet` for sets whose element membership is based on `equals` method. This specification is abstract and uses an Alloy set to represent the set of elements.
- The `java.util.TreeSet` class that uses red-black trees to implement `java.util.AbstractSet`.
- A set that represents a subset view on another set; this specification also shows how a method argument that is essentially a function can be handled.

Finally, we illustrate how behavior of methods can be automatically analyzed in an interactive fashion.

3.2.1 Set interface

Consider the following example AAL specification for a part of the `java.util.Set` interface:

```
package java.util;

public interface Set extends Collection {
    //@ ensures {
    //@   \result = (o instanceof Set &&
    //@              containsAll(o) &&
    //@              o..containsAll(this))
    //@ }
    boolean equals(Object o);

    //@ ensures {
    //@   \result = all o: Object |
    //@               c..contains(o) => contains(o)
    //@ }
    boolean containsAll(Collection c);
}
```

The specifications are given only for methods `equals` and `containsAll`, because the interface `Set` should not specify all standard axioms for sets. Otherwise, subclasses such as `Subset` view could not be subtypes of `Set`, as noted earlier. We present in the example only the most relevant parts of the specification; AAL additionally borrows from JML parts that specify exceptional behaviors (when a method returns throwing an exception), access modifiers (`public`, `protected`, and `private`), inheritance of specifications etc.; for details see [19].

Since the specifications omit preconditions (there are no `requires` annotations), they are `true` by default, as in JML. The postcondition for `equals` specifies that the result is `true` iff `o` is a non-`null` object of `Set` or its subclass and `this` and `o` are subsets of each other. The method `containsAll` determines if `c` is a subset of `this`; the result is `true` iff every object contained in `c` is also contained in `this`; '`:`' denotes set membership.

3.2.2 Abstract set

Consider next the following AAL specification for a part of the abstract class `java.util.AbstractSet` that implements the `Set` interface:

```
package java.util;

abstract class AbstractSet implements Set {
    //@ model s: set Object;

    //@ invariant {
    //@   allnn disj e1, e2: s | !e1..equals(e2)
    //@ }

    //@ ensures {
    //@   \result = ( o in s ||
    //@               somenn e: s | e..equals(o) )
    //@ }
    boolean contains(Object o);

    //@ ensures {
    //@   \old(contains(o)) => s = \old(s),
    //@                        s = \old(s) + o
    //@   \result = (s != \old(s))
    //@ }
    //@ modifies { s }
    boolean add(Object o);
}
```

The *model* field `s` declares a (mathematical) set of objects; `set` is an AAL keyword, and '`: set`' declares a subset. This field is used in specifications, but it does not need to be in an implementation for this class or its subclasses [19]. The class invariant specifies that objects in the set are not equal with respect to the `equals` method. The keyword `disj` declares disjoint subsets; in this case, it ensures that `e1` and `e2` are distinct. The symbol '`!`' denotes negation.

The postcondition for `contains` specifies that the result is `true` iff `o` itself is in `s` (which handles the case when `o` is `null`) or there exists a non-`null` object in `s` that is `equals` of `o`; '`in`' denotes subset/membership operation, and '`||`' denotes disjunction.

Each of the methods presented so far is a *pure* method (also referred to as an *observer*), i.e., it does not modify the state (there is no `modifies` annotation) and its specification expresses properties of only one state. The method `add` is a *mutator* method that modifies the state and also returns a value. Its specification 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).

In AAL postconditions, as in JML, prefixing an expression with `\old` indicates its evaluation in the pre-state, e.g., `\old(s)` is the value of model field `s` in the pre-state, and `\old(contains(o))` is an invocation in the pre-state. (It is not legal to apply `\old` to the invocation of mutator methods.) The formula `F => G,H` means that if `F` holds, `G` should hold also; if not, `H` should hold. The postcondition for `add` specifies that the set is unchanged if it `contains` the object `o` in the pre-state; otherwise, `o` is added to the set in the post-state.

The `modifies` annotation in AAL is the same as in JML: it lists the fields that the method can change; the assignment to other fields is not allowed. We discuss the semantics of `modifies` in more detail in the next subsection. Intuitively, the method `add` can change only the field `s` (of `this`); all other fields retain their pre-state values.

3.2.3 Set implementation

Consider next the following AAL specification for a part of the class `java.util.TreeSet` that implements sets (of comparable elements) with red-black trees:

```
package java.util;

class TreeSet extends AbstractSet {
    Node root;

    static final boolean RED   = false;
    static final boolean BLACK = true;

    static class Node {
        Object key;
        Node left = null;
        Node right = null;
        Node parent;
        boolean color = BLACK;
    }
    //@ represents s <- root.*(left + right).key;
    //@ depends s <- root +
    //@   \fields_of(root.*(left + right));

    //@ invariant { // RedBlackFacts
    //@   // 1. every node is red or black,
    //@   //    by construction
    //@   // 2. all leafs are black,
    //@   //    by construction
    //@   // 3. red nodes have black children
    //@   allnn e: root.*(left + right) |
    //@     e.color = RED =>
    //@       e.(left + right).color in BLACK
    //@   // 4. all paths from root to leafs
    //@   //    have same number of black nodes
    //@   allnn e1, e2: root.*(left + right) |
    //@     null in e1.(left + right) &
    //@             e2.(left + right) =>
    //@       #(e1.*parent & BLACK.~color) =
    //@       #(e2.*parent & BLACK.~color)
    //@ }

    //@ invariant { // BinarySearchTreeFacts
    //@    ...
    //@ }

    //@ // inherits spec from the superclass
    boolean add(Object o) { ... }

    //@ does { \result = new Subset(this, p) }
    Set subSet(Predicate p) { ... }
}
```

Each tree has a `root` of class `Node`; each node has its data stored in `key`, pointers to its `parent` and its `left` and `right` children, and `color` which can be `RED` or `BLACK`.

The `represents` annotation specifies the *abstraction function*, i.e., how the concrete fields in objects that build a `TreeSet` relate to the model field in `Set`; '`*`' denotes reflexive transitive closure, and '`+`' denotes set union. The `depends` annotation specifies a set of the concrete fields that the model field depends on; `\fields_of` represents all fields for each object in the given set.

The semantics of `modifies` takes into account these dependencies: if a method can modify a model field `f`, then it can also modify the fields that `f` depends on. In the example, the `add` method in `TreeSet` can modify all fields in the tree, since `add` in `AbstractSet` can modify field `s`. Conceptually, the `depends` annotation allows mutator methods in `TreeSet` to inherit their specifications from `AbstractSet` without any change, thus enabling modular reasoning. The full treatment of `modifies` is beyond the scope of this paper; details can be found in [20, 29].

The class invariant specifies the structural constraints for a red-black tree. All four properties can be easily expressed in AAL; '`#`' denotes the size of the set and '`~`' denotes the transpose relation, effectively following the field backwards. This example omits the invariants for a binary search tree, such as acyclicity, ordering of elements, and a definition of the `parent` relation as the transpose of the union of `left` and `right`. These invariants are also straightforward to express in AAL [24].

The main property to check for an implementation of a method is *code conformance*, i.e., whether the code correctly implements the specification. We call a state *valid* iff all objects in the state satisfy their respective class invariants. Code conformance requires showing that for all valid pre-states that satisfy method precondition, the execution of the method body produces a post-state that is valid and satisfies the postcondition. In Alloy, this is schematically expressed as:

```
all s, s': State |
  valid(s) && pre(s) && body(s, s') =>
    valid(s') && post(s, s')
```

One way to check code conformance is to translate the code of the method body into an Alloy formula and analyze the above implication with the Alloy Analyzer. Jackson and Vaziri developed a technique for automatic translation of a subset of Java into Alloy [15]. Work proceeds on optimizing this approach [36]; although it does not yet scale, it can handle a realistic red-black tree implementation in which the checked procedures have up to 50 lines of code. Another way to check code conformance is to use automatic testing. The TestEra framework [24] generates cases automatically from invariants and has also been demonstrated on the red-black tree implementation.

The method `subSet` is used to create a subset view. The model for this method uses a constructor for `Subset` that creates a a new view on `s` with filter `p`.

3.2.4 Subset view

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

More generally, a subset view on a set can be created by giving a *predicate* (boolean returning function) that determines

membership in the subset. In a language that supports first-class functions, a view analogous to the above view in JCF could be created by `so.subSet(p)`, where `so` is a set and `p` is a predicate defined, for example, as `(lambda (o) (and (>= o fromElement) (< o toElement)))` in Scheme. In Java, predicates, and first-class functions in general, are obtained by wrapping them within objects.

The following is a sample AAL annotated abstract class for predicates:

```
abstract class Predicate {
    //@ model def: set Object;

    //@ ensures { \result = o in def }
    boolean admits(Object o);
}
```

Each `Predicate` object has a model field `def` that denotes a (mathematical) set of objects (extensionally) defining the predicate. This field contains the *admissible* objects, i.e., the objects for which the functional predicate would evaluate to `true`. The method `admits` determines if the input object `o` is admissible for `this` predicate.

The following is an AAL annotated class for subset views:

```
class Subset implements Set {
    Set on;
    Predicate filter;

    //@ invariant {
    //@   allnn ss: *on | ss !in ss.^on // acyclic
    //@   null !in on + filter
    //@ }

    //@ requires { null !in s + p }
    //@ ensures { on = s && filter = p }
    //@ modifies { on + filter }
    Subset(Set s, Predicate p) { ... }

    //@ ensures {
    //@   \result = (filter..admits(o) &&
    //@              on..contains(o))
    //@ }
    boolean contains(Object o) { ... }

    //@ does {
    //@   filter..admits(o) => on..add(o),
    //@                        modifies {}
    //@ }
    boolean add(Object o) { ... }
}
```

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

The class invariant requires that subset views have no cyclic dependencies along `on` fields, i.e., that for each subset view, there be (transitively) a backing set that is not a view; the operator '`^`' denotes transitive closure.

The constructor creates a `Subset` from the given set and predicate. The precondition requires that these arguments be non-`null`. The method `contains` determines the membership of an object in a subset view. The postcondition requires that the input object `o` be 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 in the state. Using `modifies` as a subformula allows the user to specify conditional changes easier than by putting all fields that can change in the `modifies` clause and listing that they remain the same when some condition holds.

3.2.5 Analysis

We next present an example analysis of the above classes. The analysis focuses on a simple property of the behavior of the `add` methods. We show how to interactively use the analysis results to strengthen some assumptions in the specifications and to recheck the new specifications.

Consider the following annotated code for checking addition:

```
TreeSet ts; Subset su; Object o;
...
//@ assume { null !in ts + su }
ts.add(o);
//@ assert { su..contains(o) }
```

The `assume` annotation specifies the properties that the code can expect to hold: `ts` and `su` are non-`null` objects. The `assert` annotation specifies the properties that the code is expected to establish at a certain control point. The above code sequence has only one (mutator) method invocation, and thus only two states. It requires that an addition of an object `o` to a set `ts` also make `o` a member of a subset `su`. Additionally, the method can assume that the pre-state is valid (all objects in the pre-state satisfy their class invariants), and the method body has to ensure that the post-state is also valid.

The Alloy Analyzer checks (the translation into Alloy of) the above sequence and produces a counterexample, as expected, because `su` is not constrained to be a subset view on `ts`. We next add this constraint to our property:

```
//@ assume {
//@   null !in ts + su
//@   su.on = ts
//@ }
ts.add(o);
//@ assert { su..contains(o) }
```

The Alloy Analyzer checks the new sequence and once again produces a counterexample. This time, the assertion fails because the predicate of `su` does not admit `o`. We next add a constraint to rule this case out, too:

```
//@ assume {
//@   null !in ts + su
//@   su.on = ts
//@   su.filter..admits(o)
//@ }
ts.add(o);
//@ assert { su..contains(o) }
```

The Alloy Analyzer checks the new sequence and once again produces a counterexample. This time, the assertion is not

established because `ts` contains itself as an element. In this case, invoking `ts..contains(ts)` (i.e., `ts..equals(ts)`) is the culprit. In Java, it results in an infinite recursion; in AAL, it results in an underspecified relation as we explain in Section 5.4. We rule out such instances by modifying the class invariant for `AbstractSet` so that a set does not contain itself (transitively through other `AbstractSet`s):

```
class AbstractSet {
    //@ invariant {
    //@   allnn disj e1, e2: s | !e1..equals(e2)
    //@   this !in ^s
    //@ }
}
```

The Alloy Analyzer again checks the sequence and once again produces a counterexample. This time, the assertion fails because `ts` contains as an element a subset view on `ts`, i.e., `ts` indirectly contains itself, which leads to the above problem. One way to rule out these instances is to constrain abstract sets not to include sets as members. (The keyword `no` requires the set to be empty; '`&`' denotes set intersection.)

```
class AbstractSet {
    //@ invariant {
    //@   allnn disj e1, e2: s | !e1..equals(e2)
    //@   allnn a: AbstractSet | no a.s & Set
    //@ }
}
```

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

4 Alloy

In this section we describe the basics of Alloy; details can be found in [10, 11]. 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 *paragraph*s that can be of two kinds: signatures, used for construction of new types, and a variety of formula paragraphs, used to record constraints.

4.1 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 Object {}
```

introduces `Object` as an uninterpreted type (or a set of indivisible atoms). A signature may inherit fields and constraints from another signature. For example,

```
sig Predicate extends Object {
  def: set Object }
```

declares `Predicate` to be a subset of `Object`. The field `def` declares a relation from `Predicate` to `Object`. In a field declaration, the keyword `set` specifies that `def` maps each atom `p` in `Predicate` to a set of atoms in `Object`, i.e., `p` is an arbitrary relation; the keyword `option` specifies a partial function; omitting any keyword specifies a (total) function.

In a signature declaration, the keyword `static` specifies the declared signature to contain exactly one atom. For example, the following declares `null` to be a singleton subset of `Object`; a singleton set can be viewed as a scalar:

```
static sig null extends Object {}
```

4.2 Formula paragraphs

Properties of signature atoms are recorded as logical formulas declared using formula paragraphs. To indicate that such a property always holds, the formula is packaged as a *fact* paragraph. A property may be defined without imposing it as a permanent constraint by packaging it as a *function* paragraph to be applied elsewhere. A theorem about a specification or a property that is intended to hold is packaged as an *assertion* paragraph; a tool can check an assertion by searching for a counterexample, i.e., a model of the formula's negation.

4.2.1 Relational expressions

A formula paragraph is formed using Alloy expressions. The value of any expression in Alloy is always a relation—that is a collection of tuples. 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 are 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.

4.2.2 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.

4.2.3 Functions, facts and assertions

A function (`fun`) is a parametrized formula that can be applied by binding its parameters to expressions whose types match the declared parameter types. By default, a function returns a boolean value—the value of the formula in its body. A function may return a (non-boolean) relational value.

Functions can be grouped around the type of the first argument, like the methods of a class in an object-oriented language. For example, a "receiver" and a "result" argument can be declared with the following shorthand:

```
fun S::f(...): T {...}
```

This declaration is equivalent to

```
fun f(this: S, result: T, ...) {...}
```

where `this` and `result` are (reserved) keywords that name the arguments declared anonymously, and the ellipsis in the new argument list is the old argument list. Note Alloy's convention of treating the second argument in a function declaration as the function's result.

A function can also be applied with its first argument presented in prefix position; for example,

```
s..f(a, b, c)
```

is short for

```
f(s, a, b, c)
```

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.

5 Translation

In this section, we describe the translation of AAL annotations and Java code into Alloy. We organize the presentation of our translation by the constructs that AAL supports. We illustrate each construct using the views example from Section 3.

5.1 Inheritance

The translation of inheritance into Alloy is similar to compilation of OO languages, involving creation of virtual function tables. Details can be found in [25]. In outline, the translation has six steps:

1. Compute a hierarchy of class declarations.
2. Construct `sig Object` that represents the set of all objects in the heap.
3. Translate each `class` declaration into a `sig` declaration, with appropriate subset relationship.
4. Translate each method specification into an Alloy function with a unique name.

```
// class hierarchy

sig Object {}
  sig Set extends Object {}
    sig AbstractSet extends Set {}
      sig TreeSet extends AbstractSet {}
    sig Subset extends Set {}
  sig Predicate extends Object {}

static sig null extends Object {}

// distinct classes share only the null reference
fact nullIsCommon {
  AbstractSet & Subset = null
  Set & Predicate = null
  null in TreeSet }

// no objects for interface without a model field
fact noConcreteObjects {
  no Set - AbstractSet - Subset - null }

// dynamic types

fun Object::is_Object() {
  this in Object - AbstractSet -
          Subset - Predicate - null }

  fun Set::is_Set() {
    this in Set - AbstractSet - Subset - null }

    fun AbstractSet::is_AbstractSet() {
      this in AbstractSet - Subset - null }

      fun TreeSet::is_TreeSet() {
        this in TreeSet - null }

    fun Subset::is_Subset() {
      this in Subset - null }

  fun Predicate::is_Predicate() {
    this in Predicate - null }
```

Figure 1: Translation of the class hierarchy for the subset view example.

5. Add *dispatching* functions that model dynamic dispatch based on the receiver type.
6. Replace `super` with appropriate static invocation.

We illustrate these steps by translating the `equals` methods for sets presented in Section 3. Throughout, the Alloy text that results has been indented to show its correspondence to the class hierarchy.

Step 1 computes the following class hierarchy for our running example:

```
Object
 +-- AbstractSet (implements Set)
     +-- TreeSet
 +-- Subset (implements Set)
 +-- Predicate
```

Figure 1 shows the translation done by steps 2 and 3. The translation introduces `null` as an atom of `Object` that is shared by all classes. We have previously modeled `null` as the empty set, but that approach is too restrictive; for example, it disallows `null` to be in a set of objects. The current

approach uses the following semantics for `null`: dereferencing `null` along a field produces the empty set and invoking a method on `null` always results in an unsatisfiable formula, as we show in the translation of methods.

Step 4 introduces Alloy functions for the method specifications and behaviors. Since Alloy does not have dynamic dispatch, each function has to have a unique name. For example, the translation for the `equals` methods introduces the following functions:

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

  fun Set::Set_equals(o: Object): boolean {
    ... }
```

The function corresponding to the (default) `equals` method in class `Object` is built from its behavior. We show later how the translation builds the body for the function that corresponds to the specification for the `Set` interface.

Step 5 adds the dispatching function for `equals`:

```
fun Object::equals(o: Object): boolean {
  (this..is_Object() &&
   this..Object_equals(o)) ||
      (this..isAbstractSet() &&
       this..Set_equals(o)) ||
        (this..is_TreeSet() &&
         this..Set_equals(o)) ||
      (this..is_Subset() &&
       this..Set_equals(o)) ||
    (this..is_Predicate() &&
     this..Object_equals(o)) }
```

This function models dynamic dispatch based on the receiver type. It allows method invocations `o..equals(p)` from AAL to be translated into identical function applications, `o..equals(p)`, in Alloy. (Recall that '`..`' in an Alloy expression denotes function application without dynamic dispatch semantics.) Since `Predicate` does not override `equals`, it inherits `equals` from `Object`; likewise, `AbstractSet`, `TreeSet`, 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.

5.2 State

To handle mutator methods and their sequencing, the 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 methods that express properties on state(s). For our running example, the signature `State` is:

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

Each atom of `State` models a state. Each relation in `State` corresponds to one of the field declarations in the AAL specifications of the classes. (In practice, the names of the fields in `State` are fully qualified with their class names.) This translation essentially flattens out the hierarchical structure of the heap. The *multiplicity marking* '`!`' on `on` and `filter` specifies that for each atom `t` of `State`, `t.on`, respectively `t.filter`, is a total function from non-`null` `Subset` atoms to `Set` atoms.

5.3 Methods and class invariants

AAL annotations of methods and class invariants are translated into Alloy functions. We first illustrate translation of the class invariant for the `Subset` class:

```
    //@ invariant {
    //@   allnn ss: *on | ss !in ss.^on // acyclic
    //@   null !in on + filter
    //@ }
```

The translation generates the following Alloy function:

```
fun Subset::Subset_inv(t: State): boolean {
  all ss: *(t.on)[this] - null | ss !in ^(t.on)[ss]
  null !in t.(on + filter)[this] }
```

The function name is chosen according to the rules for handling inheritance, and the function has one parameter of type `State`. Each field access in the body, in this example `this.(on + filter)`, is replaced with the access of the corresponding value in the state, i.e., `t.(on + filter)[this]`. JML allows class invariants to be inherited, and thus a body for a class invariant may be, in general, conjoined with the class invariants from the superclass(es).

The translation also introduces an Alloy function that determines whether a given state is valid, i.e., whether all objects satisfy their class invariants:

```
fun State::valid(): boolean {
  all o: Object - null | o !in this.free =>
                         o..inv(this) }
```

(The `free` set of unallocated objects is introduced in Section 5.5.) The function `inv` is the dispatcher for class invariants. Like the `equals` function shown above, it wraps the functions of the individual classes, dispatching by testing membership of the receiver object in the sets that correspond to the classes.

We next illustrate translation of a pure method, using the following example from `AbstractSet`:

```
    //@ ensures {
    //@   \result = ( o in s ||
    //@               somenn e: s | e..equals(o) )
    //@ }
    boolean contains(Object o);
```

In this example, the method is annotated with a postcondition and the translation uses it to model the method. When a method has a `does` annotation, the translation can use it for the model.

The translation generates the following Alloy function:

```
fun AbstractSet::AbstractSet_contains(t: State,
      o: Object): boolean {
  result = o in t.s[this] ||
           some e: t.s[this] - null |
             e..equals(t, o) }
```

This function has, in addition to the method parameter, one `State` parameter. This parameter is also added to each invocation in the method body; each of these invocations is on a pure method, since pure methods cannot invoke mutator methods.

We next illustrate translation of a mutator method, using the following example from `AbstractSet`:

```
    //@ ensures {
    //@   \old(contains(o)) => s = \old(s),
    //@                        s = \old(s) + o
    //@   \result = (s != \old(s))
    //@ }
    //@ modifies { s }
    boolean add(Object o);
```

Again, the translation uses the postcondition to model the method, but it additionally uses the `modifies` annotation, because the method is a mutator. For a mutator method, the translation can also use the `does` annotation, which can include `modifies`.

The translation generates the following Alloy function:

```
fun AbstractSet::AbstractSet_add(t, t': State,
      o: Object): boolean {
  t.s[this]..contains(t, o) =>
    t'.s[this] = t.s[this],
    t'.s[this] = t.s[this] + o
  result = (t'.s[this] != t.s[this])
  modifiesFields_s(t, t', this)
  modifies(t, t', this) }
```

This function has, in addition to the method parameter, two `State` parameters, for pre-state `t` and post-state `t'`. We use primed notation to signify post-state; the prime mark has no built-in semantic significance in Alloy. The expressions prefixed with `\old` are translated to evaluate in the pre-state. The field accesses and the pure method invocation in the body of this general method are translated as illustrated earlier.

The `modifies` annotation, which specifies the modified fields, is translated into several function applications. 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 frame condition `modifiesFields_f` constrains all the fields of an object but `f` to be unchanged. If several fields are listed in the modifies clause, a composite frame condition must be generated, in the same way that a single `modifies` condition is generated for the whole set of objects that may change.

The function `modifies` is invoked with the set of objects that may be modified by the mutator method. 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, but it allows `s` to change:

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

In general, the translation adds state(s) to an expression that appears in a specification depending on the expression (an invocation of a pure method `pm`, an invocation of a mutator method `mm`, or a field access) and whether the specification is for a pure method PM (i.e., it has only one state) or for a mutator method MM (i.e., it has a pre-state and a post-state):

- Invocation `o..pm(...)` in PM or `\old(o..pm(...))` in MM becomes `o..pm(t, ...)`, where `t` is the (pre)state.
- Invocation `o..pm(...)` (not inside `\old`) in MM becomes `o..pm(t', ...)`, where `t'` is the post-state.
- Invocation `o..mm(...)` in MM becomes `o..mm(t, t', ...)`.
- Field access `o.f` in PM or `\old(o.f)` in MM becomes `t.f[o]`.
- Field access `o.f` (not inside `\old`) in MM becomes `t'.f[o]`.

5.4 Recursion

The Alloy Analyzer currently does not support recursive functions. We allow AAL specifications to be recursive. Also, the translation given above results in recursive functions (in particular through dispatching functions). We next outline the translation that eliminates recursive functions so that the models can be analyzed with the current Alloy Analyzer.

Elimination of recursion introduces new relations in the signature `State` and constrains these relations using the corresponding functions. For each set of mutually recursive functions, 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 }
```

Since `add` is a mutator method, the type of its corresponding relation has an additional (post-)state.

For example, the translation eliminates recursion from `equals` by replacing it with:

```
fact equalsFixPoint {
  all This, o: Object | all t: State |
  This..equals(t, o)
  <=>
  {
   (This..is_Object() &&
    This..Object_equals(t, o)) ||
         (This..is_TreeSet() &&
          This..Set_equals(t, o)) ||
       (This..is_Subset() &&
        This..Set_equals(t, o)) ||
     (This..is_Predicate() &&
      This..Object_equals(t, o)) } }

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

The fact `equalsFixPoint` defines the relation `equalsR` so that its tuples represent a fixed point of the `equals` methods. That is, it will contain a tuple `(s,o1,o2)` iff in state `s`, object `o1` is equal to object `o2`; '`<=>`' denotes equivalence. Note that this translation does not constrain the relation to be the least fixed point; if the definitions of recursive functions have several fixed points, AA will explore all of them. This is an appropriate approach for relational specifications: a recursive definition with several fixed points corresponds simply to an underspecified behavior and the analysis should explore all possible behaviors.

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

```
fact addFixPoint {
  all This, o: Object | all t, t': State |
  This..add(t, t', o)
  <=>
  {
     (This..is_TreeSet() &&
      This..AbstractSet_add(t, t', o)) ||
   (This..is_Subset() &&
    This..Subset_add(t, t', o)) } }

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

5.5 Object creation

AAL supports object creation in mutator methods. The translation handles it by adding 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 then allocates an object by removing it from the freeset.

The translation disallows the objects in `t.free` to be pointed to by any field. The translation also makes free objects to point to no 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 the function `modifies`:

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

Consider the following `Subset` constructor for our running example:

```
    //@ requires { null !in on + filter }
    //@ ensures { on = s && filter = p }
    //@ modifies { on + filter }
    Subset(Set s, Predicate p) { ... }
```

This constructor is translated to:

```
fun Subset_new(t, t': State,
      s: Set, p: Predicate): Subset {
  result in t.free
  result..is_Subset()
  t'.on[result] = s
  t'.filter[result] = p
  modifiesFields_on_filter(t, t', result)
  modifies(t, t', result) }
```

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` also contains the translated `modifies` clause.

5.6 Code sequences

AAL supports analysis of (loop-free) code sequences that may have method invocations. Code with loops can be translated into loop-free code using, for example, a finite unrolling of loops [4, 15].

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

```
TreeSet ts; Subset su; Predicate p; Object o;
...
//@ assume { null !in ts + su + p }
if (p.admits(o)) {
    su.remove(o);
    //@ assert { !ts..contains(o) }
}
ts.add(o);
//@ assert { p..admits(o) => su..contains(o) }
```

The translation uses *verification condition generation*, similar to that of ESC [4], and generates an Alloy assertion that threads state through the sequence as shown in Figure 2. This translation requires all (post-)states to be `valid`. If methods are already checked to be correct implementations, then this requirement can be omitted.

Note that all local variables declared in this example do not change their values in the sequence and thus can be universally quantified. In general, a local variable can be bound by several assignment statements in a code sequence, so it cannot be translated simply to a quantified variable. One can handle this by treating local variables as "fields" of the state. Alternatively, instances of the same local variable can be given distinct names, putting the code into single static assignment (SSA) form [2]. This tends to result in smaller Alloy formulas and thus faster analysis.

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

assert {
  all ts: TreeSet, su: Subset,
      p: Predicate, o: Object {
    null !in ts + su + p && valid(t0) =>
      p != null &&
      (p..admits(t0, o) =>
        su != null &&
        (su..remove(t0, t1, o) =>
          valid(t1) &&
          !ts..contains(t1, o) &&
          (ts..add(t1, t2, o) =>
            valid(t2) &&
            p != null && su != null &&
            p..admits(t2, o) => su..contains(t2, o)
          )
        ),
        ts != null &&
        (ts..add(t0, t1, o) =>
          valid(t2) &&
          p != null && su != null &&
          p..admits(t1, o) => su..contains(t1, o)
        )
      ) } }
```

Figure 2: Translation of the example code sequence.

6 Related work

Recent annotation languages, such as the Java Modeling Language (JML) [19] and the assertion subset of Eiffel [26], are themselves based on much earlier foundational work in program annotation. ANNA [22], for example, was an early annotation language for Ada; the Larch interface languages, such as Larch/C++ [18] led the way for modeling of the heap in an object-oriented language. Eiffel's innovation was to integrate pre- and post-conditions into the programming language itself, and to apply the assertions of superclasses automatically to their subclasses. A systematic analysis of the relationship between the assertions of classes at different levels in the class hierarchy has been developed recently by Findler and Felleisen [6]. Eiffel typifies the languages, like JML, that build on the expression language of the programming language; ANNA and Larch/C++ typify those languages, like AAL, that build assertions on a more abstract semantics.

Our work builds on these foundations. Its novelty is twofold: using a relational semantics of the heap, and providing opportunities for automatic compile-time analysis based on exhaustive search. These aspects are intertwined: our analysis technology relies on the uniform representation of data structures as relations, which are easily encoded as boolean matrices, making possible the application of SAT solving technology. The use of relations as a modeling tool for data structures is not new; it goes back to the earliest work on data abstraction [5], and to the entire field of semantic data modeling. Specification and assertion languages for object-oriented programs have, however, invariably preferred algebraic semantics instead, despite the use of graphs in shape analysis as abstractions of the heap.

To our knowledge, our work is unique in offering the prospect of fully automatic compile-time analysis for annotations that can capture complex structural properties. Several other annotation languages that are equally rich (or richer) have been developed, such as JML and the assertion subset of Eiffel, but these have been designed primarily for run-time checking.

Static analyzers, such as the Extended Static Checker [4], the Three-Valued-Logic Analyzer [21, 31], the Pointer Assertion Logic Engine [27], and Role Analysis [17], are automatic, but cannot handle such expressive annotations. None of these, to our knowledge, for example, can express and analyze the red-black invariants (at least without additional manual instrumentation).

Our analysis would be termed *unsound* in the program analysis literature: that is, it may fail to find a counterexample if one exists. Since we view its purpose as the discovery of flaws rather than the demonstration of correctness, we prefer to think of it as *incomplete*: not all bugs are found, but it is sound in the sense that any counterexample reported will represent a legitimate execution. None of the analyses mentioned above reliably produce sound counterexamples. In fact, most are also not complete: the authors of ESC have argued convincingly that making their checker complete would cause it to generate many more false alarms.

The LOOP project [35] aims at proving properties of Java classes by translating them and their JML specifications into logical theories in higher order logic. A theorem prover is then 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 [8].

Our analysis translates AAL annotations and Java code into Alloy. Alloy has been previously used for object-oriented modeling. Two previous works propose encodings of the heap in Alloy: one, like ours, models references directly as relations [12]; the other [13] represents mutations as changes in the mapping of references to mathematical objects. This second approach is more modular, but results in a less tractable analysis. AAL builds on these approaches by adding direct support for (single) dynamic dispatch and recursion, and by setting these constructs in the context of an annotation language.

The approach to checking code against partial specifications that we advocate in this paper is described in more detail elsewhere [15] and is the subject of ongoing work [36]. Although it has yet to be shown to scale, it has been applied successfully to a practical implementation of a red-black tree.

Marinov and Khurshid developed TestEra [24], an Alloy-based framework for automated testing of Java programs. In TestEra, specifications are written in Alloy, and the Alloy Analyzer is used both as an automatic test case generator and as a test oracle. TestEra discovered bugs in a previous version of the Alloy Analyzer. TestEra has also been used to systematically test data structures, such as red-black trees.

Formal specification languages have been also extended with object-oriented features for greater expressiveness and for

more direct modeling of object-oriented systems. Object-Z [32], for example, extends the Z specification language [33] with a new kind of schema in which operations may be associated with instances of a class. Object-Z is not used as an annotation language, and is not amenable to automatic analysis.

Objects and inheritance have also been added to declarative languages. For example, Prolog++ [28] extends Prolog. OOLP+ [3] aims to integrate object-oriented paradigm with logic programming by translating OOLP+ code into Prolog without meta-interpretation.

Keidar et al. [16] add inheritance to the IOA language [23] 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. AAL allows subclasses that arbitrarily change the behavior of inherited methods, as illustrated with the `equals` methods.

7 Conclusions

The key idea in this paper is that a simple logic with relational operators suffices as a semantic basis for an annotation language. The choice of this logic was motivated by the desire for fully automatic analysis of object-oriented programs. The logic, due to its relational operators, has a direct appeal from a specification viewpoint also. Navigation expressions (especially those involving transitive closure) are succinct and easy to write; witness the red-black tree invariants of Section 3.

The analysis upon which our scheme relies has two properties that make it well suited to analysis of object-oriented programs. First, a huge space of cases can be covered; the analysis therefore tends to find subtle problems much more readily than testing. Second, when a claim does not hold, the analysis produces a counterexample: one or more configurations of the heap that show why the claim is invalid.

Our examples have shown that this approach can be applied to program structures that are known to be a cause of complexity and errors. Currently, our translations are done systematically but manually; the analysis is performed fully automatically by the Alloy Analyzer (and counterexamples are displayed using its visualization facility). We are now implementing the translation outlined in this paper in order to build a comprehensive tool. We expect that the tool will work well for modular analysis of even quite complex classes; how well it scales for analyses amongst classes, and whether it will be economical enough for everyday use, remains to be seen.

Acknowledgments

We would like to thank Matthias Felleisen, Viktor Kuncak, and Gregory Sullivan for discussions on AAL and 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] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] D. R. Chase, M. Wegman, and F. K. 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.

[3] M. Dalal and D. 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, Dec. 1989.

[4] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

[5] J. Earley. Toward an understanding of data structures. *Communications of the ACM*, 14(10):617–627, 1971.

[6] R. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. ACM SIGPLAN 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–15, Oct. 2001.

[7] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proc. of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–180, Ottawa, Canada, 1990.

[8] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. *Software Tools for Technology Transfer*, 2001.

[9] D. Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, November 2000.

[10] D. Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. Available online: http://sdg.lcs.mit.edu/alloy/book.pdf.

[11] D. Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), Apr. 2002.

[12] D. Jackson. Object models as heap invariants. In C. Morgan and A. McIver, editors, *Essays on Programming Methodology*. Springer Verlag, 2002. (to appear).

[13] D. Jackson and A. Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, Oct. 2001.

[14] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Vienna, Austria, Sept. 2001.

[15] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, Aug. 2000.

[16] I. Keidar, R. Khazan, N. Lynch, and A. 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.

[17] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Portland, OR, Jan. 2002.

[18] G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, 1996.

[19] G. T. Leavens, A. L. Baker, and C. 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).

[20] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proc. ACM SIGPLAN 1998 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 33, pages 144–153, Oct. 1998.

[21] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symposium*, Santa Barbara, CA, June 2000.

[22] D. C. Luckham and F. von Henke. An overview of Anna, a specification language for Ada. In *IEEE Software*, volume 2, pages 9–23, Mar. 1985.

[23] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[24] D. Marinov and S. 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, Nov. 2001.

[25] D. Marinov and S. Khurshid. VAlloy: Virtual functions meet a relational language. In *Proc. Formal Methods Europe (FME)*, Copenhagen, Denmark, July 2002.

[26] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, N.Y., 1992.

[27] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.

[28] C. Moss. *Prolog++ The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.

[29] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. Technical Report 02-02, Iowa State University, Feb. 2002.

[30] M. 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`.

[31] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.

[32] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[33] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[34] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. `http://java.sun.com/j2se/1.3/docs/api/`.

[35] J. van den Berg and B. 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, Apr. 2001.

[36] M. Vaziri. Finding bugs with a constraint solver. PhD Thesis Proposal, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2002.