Object Models, Heaps, and Interpretations

Martin Rinard and Viktor Kuncak

{rinard,vkuncak}@lcs.mit.edu

January 16, 2001

Abstract

This paper explores the use of object models for specifying verifiable heap invariants. We define a simple language based on sets and relations and illustrate its use through examples. We give formal semantics of the language by translation into predicate calculus and interpretation of predicates in terms of objects and references in the program heap.

1 Introduction

Program analysis systems have traditionally operated on the source code alone, utilizing no other information about the program, its design, or its intended behavior. This report describes our initial steps towards integrating design information into a program analysis system. The expected result is a more precise, efficient program analysis system and a system that guarantees that the program faithfully implements its design. This guarantee will allow developers to rely on the design as a valid source of information about the program, integrating the design effectively across the entire lifecycle of the software. In the absence of such a guarantee, the design and implementation diverge, with the design becoming increasingly less relevant as development proceeds into the implementation and maintenance phases.

We propose to use object models as our source of design information [10, 6]. Object models are widely used and have a graphical representation that many developers find convenient. In our interpretation, they provide crucial information about the intended structure of the heap. Our interpretation therefore allows us to precisely capture a key aspect of the design (heap properties) and serves as a foundation for an analysis that can verify that the program preserves key information representation properties.

We contrast our approach with that of traditional type systems. In a traditional type system, each object has the same type throughout its entire lifetime. Type systems therefore fail to capture the different roles that a typical object may play during its lifetime in the program. We believe that we can capture these changes by further characterizing objects as follows:

- **Content-Based Classification:** The values stored in an object often determine the role that it plays.
- **Relative Classification:** The object's points-to relationships often determine its role. Specifically, objects often move between data structures as they play different roles in the computation.
- **Experience-Based Classification:** An object's lifetime is often characterized by a sequence of operations that the program performs on it. Role changes often correlate with the intiation or completion of specific operations.

Because object models focus on the relationships between objects, they are well suited for specifying relative classifications. Because they may specify information about the values that objects contain, they are also appropriate for specifying content-based classifications. Because they focus on invariants that are always valid, we view them as less well suited for specifying experience-based classifications.

We view object models as containing three basic concepts: sets, relations between sets, and properties that the sets and relations satisfy. Object models therefore describe the heap at a higher level of abstraction than the objects, values, and references present in the heap. The key to our approach is an *interpretation* that bridges the abstraction gap between the object model and the heap. Specifically, the interpretation tells how the objects, values, and references in the heap map onto the sets and relations of the object model.

2 Object Models

Researchers have developed several notations for expressing object models, among them UML [10] and Alloy [6]. Our approach is neutral — as long as the object model has the concepts of sets of objects and relations between these sets, our approach is valid. We next describe a simple object model language inspired by Alloy. We view Alloy as basically as a syntactic sugar for first-order predicate calculus with transitive closure. Figure 1 presents a grammar for the example object model language we use in this paper.

We place non-terminals such as `Def` in brackets, while terminals such as `RelID` have no brackets.

2.1 Object Models in Use

In this section we illustrate how object models can be used to refer to parts of the heap and specify constraints.

A simple way of using sets and relations are **class diagram** specifications.

```
<Def>     ::= RelID := <RelExpr> | SetID := <SetExpr> | <Constr>
<SetExpr> ::= SetID | {} | _ | <SetExpr> <SetOper> <SetExpr>
            | <SetExpr> . <RelExpr> | <RelExpr> . <SetExpr>
<SetOper> ::= + | - | &
<RelExpr> ::= RelID | <RelExpr> <RelOper> <RelExpr>
            | <RelExpr> . <RelExpr> | <RelExpr> *
            | <SetExpr> # <SetExpr>
<Constr>  ::= <SetExpr> : <SetExpr> | <SetExpr> = <SetExpr>
            | <RelExpr> : <RelExpr> | <RelExpr> = <RelExpr>
            | <RelExpr> : <SetExpr>
            | <RelExpr> :: <SetExpr> <Card> -> <Card> <SetExpr>
<Card>    ::= ! | ? | + | *
```

Figure 1: Grammar for Object Model Language

Example. Consider the following class declarations.

```
class CarList {
  Car     car;
  CarList next;
}
class Car {
  String name;
  int    id;
}
```

We can represent these relationships by the following specification.

```
next : CarList # CarList
car  : CarList # Car
name : Car     # String
id   : Car     # int
```

The notation `f : A # B` introduces a *relation* f from set A to set B. To specify that a relation is *function*, we use **cardinality constraints**. The notation `f :: A ->! B` specifies that for every element of A there is precisely one related element of B. To specify *partial functions* we use the notation `f :: A ->? B` meaning that for every element of A there is at most one related element of B. Dually, we write e.g. `f :: A !-> B` if for every element of B there is precisely one element of A which is related to it.

In the previous example we used sets to denote classes. Such sets are called *primitive sets*. It is often useful to specify a particular **subset** of a set of objects which are instances of a class. We write $D : C$ to indicate that set D is subset of set C.

Example. Consider class declarations from previous example. To refer to a subset of all possible strings which includes only strings used to name cars in

the program as opposed to all possible strings, we define the set `CarName` by writing

```
CarNames := Car.name
```

Similarly, to denote the set of only those cars that are stored in a list we use expression `CarList.car`. Let `f : A # B`. The notation `A.f` denotes set of all elements for which there exists an element of `A` related to it by the relation `f` (direct relation image). Dually, we consider `f.B` to be the set of all elements that are related to some element of `B` (inverse image of a relation). Clearly, `A.f : B`.

Example. Inverse relation image is useful for **content-based classification**. Consider the class `Owner` with a local boolean field `localResident`.

```
class Owner {
  String firstName;
  String lastName;
  boolean localResident;
}
```

Owners which are local residents are then defined by

```
LocalOwner := localResident.true
```

and those that are not by `NonlocalOwner = localResident.false`. Here, we use `false` and `true` as primitive singleton sets whose interpretation is known.

We permit set **operations** union (`+`), intersection (`&`) and set difference (`-`) on sets and relations. The symbol `_` denotes universal set and `{}` empty set.

Example. In the previous example, the set `LastLocal` of last names of local residents is defined by

```
LastLocal := Owner.lastName & localResident.true
```

whereas the set of last names which are not also first names is

```
LastOnly := Owner.lastName - Owner.firstName
```

To refer to set of elements reachable by following some field unknown number of times, we use **regular expressions**.

Example. Let local variables `owned` and `sold` be pointers to lists of cars.

```
class CarList {
  Car     car;
  CarList next;
}
CarList owned;
CarList sold;
```

The content of the list `owned` is given by the regular expression

```
owned.next*.car
```

and the content of the list list `sold` is given by `sold.next*.car`. Regular expressions denote relations defined in terms of relation composition, union of relations and transitive closure. Define relation `nodesof := next*`. Then

```
owned.nodesof & sold.nodesof = {}
```

is a constraint indicating that nodes of these lists are disjoint. To specify that objects in these lists are disjoint, we define `contentof := nodesof.car` and require

```
owned.contentof & sold.contentof = {}
```

Example. Consider definition of binary tree.

```
class TreeNode {
  TreeNode left;
  TreeNode right;
}
TreeNode t;
```

The set `(t.left*) & left.null` denotes the leftmost leaf of the tree `t`. It is defined as the intersection of nodes reachable along the `left` references from the root and nodes that have `left` pointer `null`. The set of all leaves of tree t is given by

```
t.(left+right)* & left.null & right.null
```

If we want to define a relation specifying the notion of a leaf of a tree, we lift the sets `left.null` and `right.null` to relations and use similar definition.

```
leafof = (left+right)* & _#(left.null) & _#(right.null)
```

Now we can reuse the definition of this relation to refer to sets of leaves of trees rooted at different nodes.

Next, observe that the following information for class diagram of a tree

```
left  :  TreeNode # TreeNode
right :  TreeNode # TreeNode
left  :: TreeNode ->? TreeNode
right :: TreeNode ->? TreeNode
```

does not fully specify a tree since it does not imply that all branches have disjoint nodes. We can use cardinality constraints to convey this information.

```
(left+right) :: TreeNode ?-> TreeNode
```

This illustrates usefullness of cardinality constraints on arbitrary relations.

2.2 Object Model Semantics

We provide the semantics for object models by specifying a translation from the object model syntax into closed formulas of predicate calculus with transitive closure. Sets are translated to unary predicates; set operations are naturally translated by extending logical operations to unary predicates.

Definition 1 $(\forall x)\neg[\![\{\}]\!](x)$

Definition 2 $(\forall x)[\![_]\!](x)$

Definition 3 $(\forall x)[\![S\texttt{+}T]\!](x) \iff ([\![S]\!](x) \vee [\![T]\!](x)$

Definition 4 $(\forall x)[\![S\texttt{-}T]\!](x) \iff ([\![S]\!](x) \wedge \neg[\![T]\!](x))$

Definition 5 $(\forall x)[\![S\texttt{\&}T]\!](x) \iff ([\![S]\!](x) \wedge [\![T]\!](x))$

Relations are translated into binary predicates. Full relation between sets A and B is Cartesian product $A\texttt{\#}B$.

Definition 6 $(\forall x, y)[\![A\texttt{\#}B]\!](x, y) \iff A(x) \wedge B(y)$

Set operations on relations are also defined naturally.

Definition 7 $(\forall x, y)[\![s\texttt{+}t]\!](x, y) \iff ([\![s]\!](x, y) \vee [\![t]\!](x, y)$

Definition 8 $(\forall x, y)[\![s\texttt{-}t]\!](x, y) \iff ([\![s]\!](x, y) \wedge \neg[\![t]\!](x, y))$

Definition 9 $(\forall x, y)[\![s\texttt{\&}t]\!](x, y) \iff ([\![s]\!](x, y) \wedge [\![t]\!](x, y))$

Next we define relation composition as well as direct and inverse images of sets under relations.

Definition 10 $[\![f.g]\!](x, z) \iff (\exists y)([\![f]\!](x, y) \wedge [\![g]\!](y, z))$

Definition 11 $[\![S.g]\!](z) \iff (\exists y)([\![S]\!](y) \wedge [\![g]\!](y, z))$

Definition 12 $[\![f.T]\!](x) \iff (\exists y)([\![f]\!](x, y) \wedge [\![T]\!](y))$

Transitive closure is defined as usual.

Definition 13 $(\forall x)[\![\ f]\!](x, y) \iff [\![f]\!](y, x)$

Definition 14 $(\forall x, y)[\![f*]\!](x, y) \iff [\![f]\!]^*(x, y)$ *where* * *is transitive closure operator in predicate calculus i.e.*
$(\forall x, y) f^*(x, y) \iff (\exists n \geq 0)(\exists z_1, \ldots, z_n)(f(x, z_1) \wedge f(z_1, z_2) \wedge \cdots \wedge f(z_n, y))$

Notation : denotes subset for sets and relations.

Definition 15 $[\![S : T]\!] \iff (\forall x)([\![S]\!](x) \Rightarrow [\![T]\!](x))$

Definition 16 $[\![f : g]\!] \iff (\forall x, y)([\![f]\!](x, y) \Rightarrow [\![g]\!](x, y))$

Equality of sets and relations is a useful syntactic sugar for two subset specifications. Definitional equality (`:=`) is semantically equality as well. (The purpose of distinguishing `=` from `:=` is to indicate which sets and relations are primitive and which are derived.)

Definition 17 $[\![S\texttt{=}T]\!] \iff [\![S : T]\!] \wedge [\![S : T]\!]$

Definition 18 $[\![f\texttt{=}g]\!] \iff [\![f : g]\!] \wedge [\![g : f]\!]$

Cardinality specifications are a useful way of writing heap invariants. They give upper and lower bounds on the number of elements of the source set that are related to the target set.

Definition 19
`r :: A -> [1..q] B` $\Rightarrow (\forall x)(A(x) \Rightarrow (\exists y)(B(y) \wedge r(x,y)))$
`r :: A -> [p..0] B` $\Rightarrow (\forall x)(A(x) \Rightarrow \neg(\exists y)(B(y) \wedge r(x,y)))$
`r :: A -> [p..1] B`
$\Rightarrow (\forall x)(A(x) \Rightarrow ((B(y) \wedge r(x,y) \wedge B(z) \wedge r(x,z)) \Rightarrow y = z))$

Following abbreviations are used: $+ = [1..*]$, $? = [0..1]$, $! = [1..1]$, $0 = [0..0]$. 0 as lower bound implies no constraints as well as $*$ as upper bound. Notation $r : A[x..y] \rightarrow B$ is a shorthand for $r : B \rightarrow [x..y]A$; both upper and lower bounds can be stated in the same declaration with obvious meaning.

2.3 Primitive and Derived Sets and Relations

Note that there are conceptually two kinds of sets and relations in our object models: primitive sets and relations and derived sets and relations. Each set `SetID` with a definition `SetID := <SetExpr>` is a derived set; each relation `RelID` with a definition `RelID := <RelExpr>` is a derived relation. All other sets and relations are primitive. As we will see in Section 4, once we provide a meaning for the primitive sets and relations, the definitions provide a meaning for the derived sets and relations. Primitive and derived sets and relations are translated into primitive and derived unary and binary predicates, respectively. To ensure that the object model is well-formed, we require that it have no circular definitions.

3 Heaps

We view the heap as a directed graph with labelled edges. The nodes of the graph are objects; the edges are the references between nodes. There is an edge in the graph from object o_1 to object o_2 labelled by f iff o_1 has a field named f and $o_1.f = o_2$. Each node may also have several fields with primitive values.

4 Interpretation

We establish the connection between the object model and the heap by interpreting the predicate calculus reading of the object model in terms of the heap. Thus we treat the heap as a model of the first-order theory defined by the object model. The equality symbol is interpreted as equality of nodes in the heap, in accordance with the view that every object has unique identity.

Formally we need to provide an interpretation for all predicate symbols that appear in formulas describing an object model. However, given an interpretation of primitive unary and primitive binary predicates, there is a unique way to extend that finite domain interpretation so that formulas defining derived predicates are true. To define an interpretation for an arbitrary object model, it is therefore enough to provide an interpretation for the primitive sets and primitive relations of the model. The designer and/or programmer provides this interpretation by mapping each primitive set in the object model to a corresponding set of objects and each primitive relation in the object model to a relation on objects. Typically, primitive binary relations will be mapped to edges in the heap.

There are several ways to define the mappings for primitive sets:

- **Class-Based Mappings:** A set is defined to be all objects of a given class.
- **Content-Based Mappings:** A set is defined to be all objects whose primitive fields satisfy a given property.
- **Collection-Based Mappings:** A set is defined to be all objects in a given collection.

There are also several ways to define the mappings for primitive relations:

- **Edge-Based Mappings:** A primitive relation is defined to correspond to a set of edges between two sets of objects in the heap. A typical edge-based mapping would be the set of all pairs of objects $\langle o_1, o_2 \rangle$ such that the field `f` of o_1 refers to o_2.
- **Collection-Based Mappings:** Many collection classes in standard libraries implement relations between objects. A primitive relation could correspond to the relation implemented by an instance of such a collection class.

Note that, in general, these mappings will need to refer to program variables. For any program state, mapping specification defines values of sets and relations in that state. Thus the interpretation of primitive sets and relations as well as interpretation of derived sets and relations changes over the time.

Note that we can use the concepts of derived sets and derived relations in the object model to define more sophisticated sets and relations:

- **Upstream Relative Classification:** A set is defined to be all objects in the image of a known relation on a known set. In practice, upstream relative mappings usually correspond to classifying objects into sets based on what kind of objects point to them.
- **Downstream Relative Classification:** A set is defined to be all objects in the inverse image of a known relation on a known set. In practice, downstream relative mappings usually correspond to classifying objects into sets based on what kind of objects they point to.
- **Combined Classification:** A set is defined using union, intersection, or other set operation.

Given interpretation of primitive sets and relations, it is possible to evaluate the predicate calculus reading of the object model in the current program state to determine if the heap satisfies the conditions in the object model. If so, we say that the heap conforms to the object model under the given interpretation. We say that a program conforms to an object model under given interpretation rules if and only if all of the heaps that it may build conform to the object model under interpretation given by the rules.

5 Related Work

Software specifications based on object models are widely used. The standardization of an informal approach to specifications resulted in Unified Modeling Language [10]. It is essential for our purpose to have precise formal semantics of object models to integrate design information into a program analysis system. Semantics can be given using formalisms based on predicate calculus, such as Alloy [6]. Alloy has full expressive power of first-order predicate calculus with transitive closure as well as many syntactic shorthands. Specifications in Alloy are checked on finite models of chosen size using translation into propositional calculus. In contrast, we expect to arrive at a proper subset of first-order predicate calculus so that a static program analysis is capable of producing sound even if conservative results. The original purpose of Alloy is checking of invariants of abstract program models (which are usually much smaller than actual programs). We aim at program analysis establishing conformance of actual code with a given model and view checking of abstract models as an orthogonal activity. The use of Alloy-like language for specifying program invariants is discused in [7], which also interprets predicate calculus in terms of program heap. In our example language we decided to decompose relation declaration into subset declaration, ability to form Cartesian product, and cardinality constraints. The result is that cardinality constraints can be applied to derived relations which permits aliasing properties such as linearity to be expressed easily. [7] also considers graphical notation which is not subject of this report.

Interpreting object models in terms of the heap results in specifications similar to those used in shape analysis [11]. The notion of *core predicates* in [11] is

similar to our primitive sets and relations. However, both [7] and [11] assume direct correspondence of predicate calculus formulas (or sets and relations) with nodes and edges in heap, whereas we permit more complex mapping specification.

In [9], the authors use monadic second-order logic to specify and verify heap invariants as pre-conditions, post-conditions and loop invariants by using decision procedure of MONA [8].

Shape types [4], based on graph grammars [3], can be used for similar purpose. A system for program analysis with additional information about data structures is proposed in [5]. A type-based approach for conveying alias properties is given in [12]. Documenting alias information in programs and verifying ownership properties is discussed in [2] and [1].

6 Conclusions

We introduced a small object-modeling language. We defined its syntax and semantics in terms of program heaps. We can summarize the concept of program conforming to an object model as follows:

1. an object model consists of definitions and constraints
2. definitions and constraints are translated into a finite set of formulas of a first-order predicate calculus with equality and transitive closure
3. predicate calculus formulas are interpreted in terms of the program heap
4. a mapping specifies rules that in any given program state assign meaning to primitive sets and relations
5. given a meaning of primitive sets and relations, predicate calculus formulas corresponding to object model definitions uniquely determine the meaning of derived sets and relations
6. given a meaning of primitive and derived sets of relations, the truth value of each formula corresponding to an object model specification can be evaluated
7. the program conforms to an object model if all formulas of the object model are true at a chosen set of program execution points.

The next step is to precisely define a mapping language for primitive sets and relations, then to develop and implement a program analysis algorithm that is capable of determining if a program conforms to its object model under a given interpretation. We expect that we may need to limit the expressive power of the object model and/or the mapping language to realize this goal. We may also augment the object model and interpretation language if it becomes clear that we would like to express properties that are beyond the expressive power of the current formalism of sets and relations.

7 Acknowledgements

We would like to thank Daniel Jackson for many conversations about the concepts in this technical report.

References

[1] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, 1997.

[2] David G. Clarke, John M. Potter, and Jabes Noble. Ownership types for flexible alias protection. In *OOPSLA'98*, 1998.

[3] Pascal Fradet and Daniel Le Metayer. Structured gamma. Technical report, IRISA, No 989, 1996.

[4] Pascal Fradet and Daniel Le Metayer. Shape types. In *POPL'97*, 1997.

[5] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[6] Daniel Jackson. Alloy: A lightweight object modelling notation. Technical report, MIT Laboratory for Computer Science, No 797, 2000.

[7] Daniel Jackson. Object models as heap invariants. *Collected Papers of IFIP Working Group 2.3 on Programming Methodology, eds. Annabelle McIver and Carroll Morgan*, 2001.

[8] Nils Klarlund, Anders Moeller, and Michael I. Schwartzbach. Mona implementation secrets. In *5th International Conference on Implementation and Application of Automata CIAA*. LNCS, 2000.

[9] Anders Moeller. The pointer assertion logic engine. (submitted for publication).

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison Wesley, 1999.

[11] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *Submitted to TOPLAS*, 2000.

[12] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming. Berlin, Germany. (March 2000)*, 2000.