1

Object Models as Heap Invariants

Daniel Jackson

ABSTRACT Object models are widely used for describing structural properties of object-oriented programs, but they suffer from two problems. First, their semantics is usually unclear. Second, the textual constraints that are often used to annotate diagrams are not integrated with the diagrammatic notation itself. In this paper, we show how to interpret an object model as a heap invariant by translation to a small relational logic. With a few additional shorthands, this logic can be used for textual constraints, so that annotation is just conjunction. Our semantics is simpler than those proposed by others, and accounts for features of the object model that have not been addressed in treatments that focus on more abstract models.

Introduction

Semantic data models, developed originally for databases [1, 5, 13, 26, 29], are called 'object models' in the context of object-oriented programs, and are widely used for describing both problem and implementation structure. An object model is a better starting point for a development than a description that treats objects as interacting entities with local state, because it highlights fundamental structural properties of the problem domain, and postpones decisions about the allocation of state and function to objects. For this reason, object models lie at the core of many object-oriented methods, such as OMT [24], Fusion [3], Syntropy [4], and Catalysis [7].

Object models are good for describing implementations also, since their relational structure is a close match to the class/field structure of object-oriented programming languages such as Java. An object model, unlike a traditional specification, emphasizes sharing of objects and interactions, an aspect of design that is becoming more important because of the popularity of design patterns [9], which encourage a style of design in which programs are assembled from tightly coupled configurations of collaborating objects.

Object models are simple but surprisingly powerful. The ability to represent basic aspects of the model diagramatically is also an important attraction for many developers. Despite their popularity, however, object models are not well understood, and are often used in a way that barely

exploits their power. In many cases, a purported object model is just a class diagram—a graphical representation of the class hierarchy and field declarations of a program. As we shall see, an object model is an invariant and can express properties of the program state that are not immediately evident from the syntax of the program. An object model is also not bound by the syntactic restrictions of the programming language; it is legitimate, for example, to associate fields with Java interfaces.

Most work [2, 8, 11, 12, 18, 21] has given meaning to object models by translation to a formal specification language such as Larch [10] or Z [28]. In our opinion, this loses the key benefit of object models. An object model is a lightweight formal specification; converting it to a full-fledged formal specification language obscures its simplicity, and eliminates opportunities for analysis that take advantage of its limited expressiveness.

Instead, we have developed a tiny logic that is sufficient to express object model properties, but which is still amenable to fully automatic semantic analysis [17]. Object model diagrams are translated rather directly into the logic. Moreover, a version of the logic extended with some simple shorthands gives a textual annotation language that is compatible with the diagrammatic form. This language, which with the addition of structuring mechanisms not described here is called Alloy [16], offers the 'navigation expressions' made popular by Syntropy [4], but with a more uniform structure and simpler semantics than in other languages.

In our previous work [16], we have focused on object models that describe problem structure. In this paper, we focus on the issues that arise when object models are used to describe implementations. In this context, an object model is a heap invariant, and describes a set of legal program states. Giving meaning to such an object model obviously requires interpreting the abstract mathematical notions (of sets and relations) in terms of code notions (of classes and fields). But it also seems to require giving a slightly different semantics. It seems necessary to drop the disjointness constraints that are implicit in problem models, and to introduce a notion of 'qualified sets' that allows different contexts in which polymorphic classes are used to be distinguished.

This paper represents work in progress, and is more a collection of observations than a coherent theory. Nevertheless, we believe that it demonstrates that a much simpler notion of object modelling is possible than has previously been assumed. Recently, the Object Management Group ratified as a standard the Unified Modeling Language [25], whose object modelling notation is of unprecedented complexity. A simpler semantics may not only clarify the meaning of basic object modelling constructs, but may point the way to an object modelling language that is simpler, more expressive and easier to use.

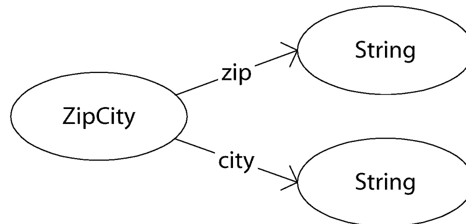
1.1 Snapshots and Object Models

The heap of an object-oriented program can be represented by a graph, with nodes for objects and edges for field references. Such a graph is called a *snapshot*, since it represents the state of the program at a particular point in time. A snapshot is usually partial, showing only the objects of certain types.

For example, a program that includes the class

```
class ZipCity {
    String zip;
    String city;
}
```

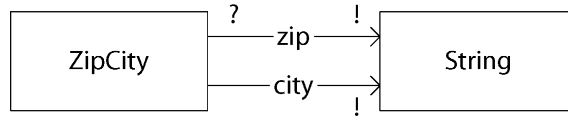
may have the following snapshot:



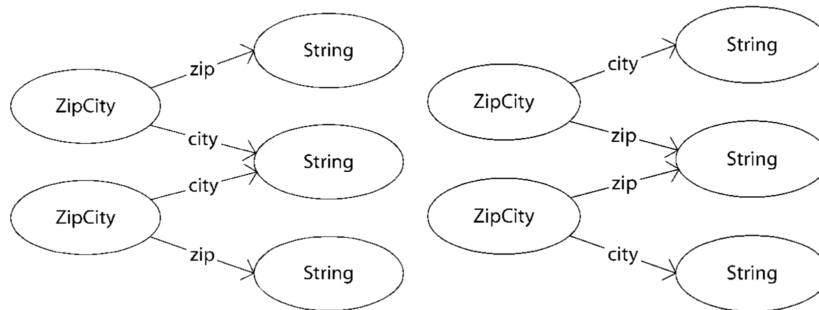
The *ZipCity* class will be used later as part of a datatype that provides a mapping from zipcodes to city names. For now, however, we are concerned only with the existence of objects and fields, and not their interpretations. In a snapshot, a node labelled C represents an object of class C; additional labels can be used to show that the object conforms to particular interfaces. An edge labelled f from m to n says that the field f of the object m holds a reference to object n. Null-valued fields are represented simply omitting the edge. (We are ignoring namespace issues, such as whether the field is accessible from outside the class, although we discuss abstraction below.)

A program typically has infinitely many snapshots. A more useful kind of diagram, the *object model*, describes an invariant: that is, a set of snapshots that are regarded as well-formed. Sometimes, the object model is merely a class diagram, showing only those properties that follow directly from the syntactic structure of the program. But it can incorporate properties that are not easily inferred from the program text, especially when textual annotations are admitted.

The following object model, for example, expresses the property that no two *ZipCity* objects have *zip* fields with the same value:



Each box in the object model represents a set of objects; arrows between boxes represent fields. The markings on the ends of the arrows indicate relative multiplicities. The exclamation mark on the head of the *zip* arrow says that each *ZipCity* object is mapped by the *zip* field to exactly one string. In other words, the *zip* field is never null. The question mark on the tail of the arrow says that each *String* object is mapped to by the *zip* field of at most one *ZipCity* object. In other words, *ZipCity* objects do not share *zip* strings. Omission of a marking implies no constraint, so the lack of a marking on the tail of the city arrow, for example, allows *ZipCity* objects to share *city* strings. The object model thus admits the snapshot on the left but not the one on the right:



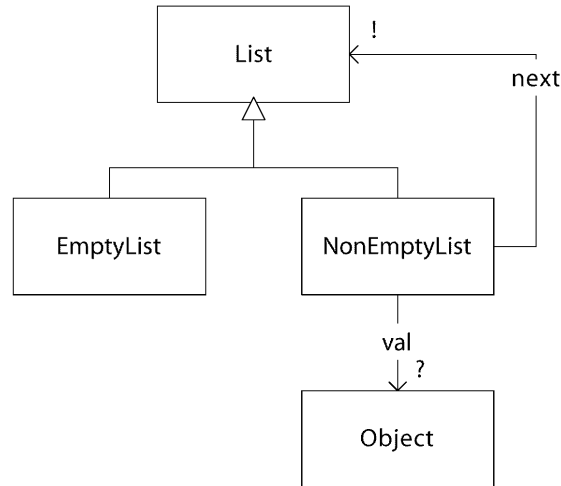
Like any invariant, an object model is a constraint with a scope limited to the fields and classes mentioned. There is no closed world assumption: the heap may contain objects of other classes, and objects may have additional fields not shown in the object model. This is important because it admits partial models that can be developed incrementally and composed with one another.

1.2 Object Model Examples

We now illustrate some additional features of object models with a series of small examples: classification of sets, shown with a linked list; instantiation of a polymorphic type, shown by using the list as the representation of a mapping from zipcodes to city names; indexed fields, shown in an array representation of the mapping; and abstract fields, shown in two more abstract views of the array representation.

1.2.1 Linked List

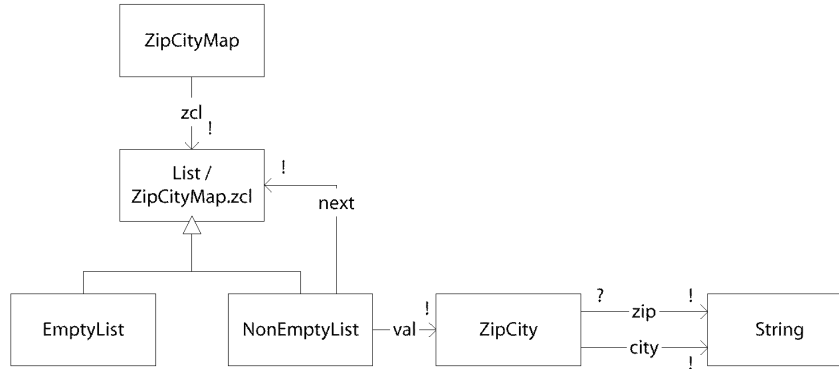
Linked lists can be represented with an abstract class and two concrete subclasses for empty and non-empty lists:



This model illustrates classification of objects. The arrow with the closed head denotes subset: each element of the set *EmptyList* is also an element of the set *List*. The sharing of the arrowhead indicates disjointness: that no object is both an *EmptyList* and a *NonEmptyList*. The multiplicity marking on the *next* field (!, meaning *exactly one*) indicates that a *NonEmptyList* always points to a *List*; the marking on the *val* field (?, meaning *zero* or *one*) allows *Lists* to contain null references as well as objects.

1.2.2 List of Records

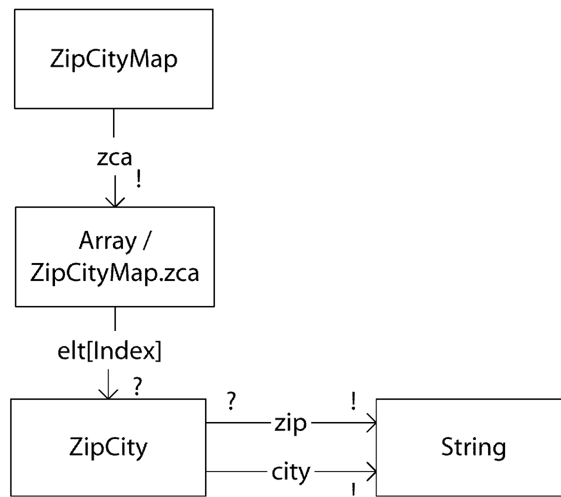
A mapping from zipcodes to city names may be implemented as a linked list of *ZipCity* objects:



The expression that qualifies the name *List* indicates that the box does not denote all *List* objects, but only those occupying the *zcl* field of *ZipCityMap* objects. This allows the object model to show that the *val* field of a *NonEmptyList* is non-null, and holds a reference to a *ZipCity* record, not an arbitrary object. The notation permits arbitrary expressions to be used to qualify sets.

1.2.3 Array of Records

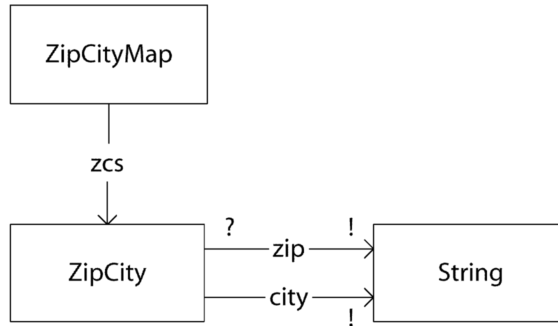
The mapping may alternatively be implemented as an array of *ZipCity* objects. The array is an object in its own right, with a field *elt*[*i*] for each index *i* mapping the array to the *i*th element. The edge labelled *elt*[*Index*] represents a collection of fields, each corresponding to one value of *Index*.



Again, the qualifier makes the *Array* box denote only those objects of class *Array* that are referenced by *ZipCityMap* objects.

1.2.4 Set of Records

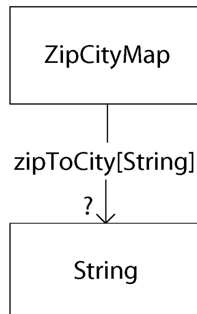
Abstracting away the array, we can show a direct association between *ZipCityMap* and its records:



In this case, the field *zcs* is an *abstract field*, and does not correspond to a field of a class. Instead, it represents part of the state of a *ZipCityMap* object that can be observed by calling its methods. The lack of a multiplicity marking on its head says that each *ZipCityMap* may be associated with any number of *ZipCity* records. The lack of a marking on the tail says that each record may belong to any number of *ZipCityMaps*.

1.2.5 Mapping

Abstracting further, we can hide the *ZipCity* record itself, using an indexed field:



The edge *zipToCity* represents a collection of fields, one for each string. For a given zipcode string, it associates a *ZipCityMap* object with at most one city name string. If zipcodes and city names were instead represented as

user-defined classes *Zip* and *City* say, the object model would show an edge labelled *zipToCity*[*Zip*] to *City*, making it more apparent that the index is the zipcode and the target of the relation the city name.

1.3 A Relational Logic

To give meaning to object models, we define a logic into which diagrams are translated, and then interpret the free variables of logical formulas in terms of programming notions.

We assume there is a universe of objects, *Object*, each of which is an uninterpreted atom. A special atom, *unit*, not in *Object*, is used to encode sets as relations, with $\{(unit, x), (unit, y), \ldots\}$ for the set $\{x, y, \ldots\}$. Scalars are represented as singleton sets; below, we use the abbreviation $set(x)$ for $\{(unit, x)\}$. Every variable in a formula therefore has a value that is a relation consisting of pairs whose first elements are drawn from the set $Object \cup \{unit\}$ and whose second elements are drawn from the set *Object*:

$$Val = \mathcal{P}\ (Object \cup \{unit\} \times Object)$$

Expressions and formulas are interpreted in an environment that assigns values to variables:

$$Env = Var \rightarrow Val$$

We shall write $\varepsilon[x/v]$ for the environment that is like ε but which binds the value x to the variable v.

To define the meaning of an expression or formula, we use two semantic functions

$$[\![\#]\!]\#: Expr,\ Env \rightarrow Val$$

which assigns a relational value to an expression in a given environment and

$$[\![\#]\!]\#: Formula,\ Env \rightarrow Bool$$

which assigns a boolean value to a formula in a given environment.

Here are the expression forms and their meanings, where p and q are arbitrary expressions, and v and w are variables, and f is a formula:

$[\![p + q]\!]\varepsilon = [\![p]\!]\varepsilon \cup [\![q]\!]\varepsilon$	*union*
$[\![p - q]\!]\varepsilon = [\![p]\!]\varepsilon \setminus [\![q]\!]\varepsilon$	*difference*
$[\![p\ \&\ q]\!]\varepsilon = [\![p]\!]\varepsilon \cap [\![q]\!]\varepsilon$	*intersection*
$[\![\sim p]\!]\varepsilon = \{(y, x) \mid (x, y) \in [\![p]\!]\varepsilon\}$	*transpose*
$[\![+p]\!]\varepsilon = [\![p]\!]\varepsilon \cup [\![p.p]\!]\varepsilon \cup [\![p.p.p]\!]\varepsilon \ldots$	*closure*

$[\![p.q]\!]\varepsilon = \{(a,c) \mid \exists b.(a,b) \in [\![p]\!]\varepsilon \wedge (b,c) \in [\![q]\!]\varepsilon\}$	*composition*
$[\![\{v,w \mid f\}]\!]\varepsilon = \{(a,b) \mid [\![f]\!]\varepsilon[set(a)/v, set(b)/w]\}$	*relation comprehension*
$[\![\{v \mid f\}]\!]\varepsilon = \{(unit,a) \mid [\![f]\!]\varepsilon[set(a)/v]\}$	*set comprehension*
$[\![v]\!]\varepsilon = \varepsilon(v)$	*variable*

The composition operator has its standard definition, but due to the encoding of sets as relations it plays several roles. When the expressions s and r represent a set and a relation respectively, the expression $s.r$ denotes the image of s under the relation r. When s and t are both sets, $\sim s.t$ is the cross-product of s and t; when these sets are scalars, $\sim s.t$ represents the pair (s,t). In practice of course, one does not write expressions in this form, but uses shorthands instead (see below).

Here are the formulas, where f and g are formulas, p and q are arbitrary expressions, and v is a variable:

$[\![p \ in \ q]\!]\varepsilon = [\![p]\!]\varepsilon \subseteq [\![q]\!]\varepsilon$	*subset*
$[\![f \&\& g]\!]\varepsilon = [\![f]\!]\varepsilon \wedge [\![g]\!]\varepsilon$	*conjunction*
$[\![!f]\!]\varepsilon = \neg [\![f]\!]\varepsilon$	*negation*
$[\![all \ v \mid f]\!]\varepsilon = \wedge\{ [\![f]\!]\varepsilon[set(a)/v] \mid a \in Object\}$	*universal quantification*

There is only one elementary formula $p \ in \ q$, written equivalently $p : q$, which is true when the relation denoted by p is a subset of the relation denoted by q (viewed as a set of pairs). Due to the encoding of scalars as sets, when x is a scalar and s is a set, $x \ in \ s$ holds when x is a member of the set s.

1.3.1 Shorthands

To make this logic practical, it must be extended with a collection of shorthands. The following are taken from our Alloy language [16]. Obviously we define the other logical operators (such as disjunction and implication), relational operators (such as * for reflexive transitive closure), and an equality operator on expressions. We let $s \rightarrow t$ be short for $\sim s.t$, so that

$$p : s \rightarrow t$$

is a formula asserting that p is a relation from the set s to the set t.

Several useful quantifiers are defined: *some* (existential quantification), the dual of *all*; *no*, the negation of *all*; and *sole* and *one* for asserting that there is at most one value, and exactly one value satisfying the formula:

$$sole \ v \mid f \equiv some \ w \mid \{v \mid f\} \ in \ w$$
$$one \ v \mid f \equiv some \ w \mid \{v \mid f\} = w$$

We use the standard shorthands for quantifier bounds, such as

$$\textit{all } v\text{: } e \mid f \equiv \textit{all } v \mid (v \textit{ in } e) \textit{ implies } f$$

Rather than defining the empty relation as a constant, we let Qe, where Q is any quantifier, be short for

$$Q\ v \mid v \textit{ in } e$$

so that *no* e, *sole* e, *some* e, *one* e mean that e contains no elements, at most one element, some element and exactly one element respectively.

1.4 Diagrams to Logic

The meaning of an object model diagram is given by translation to a formula in the logic. Features of the diagram generate constraints, which are then conjoined to give a single formula. The formula is then interpreted in terms of programming notions by binding free set variables to class names and free relation variables to field names.

1.4.1 Translating Fields and Classification

For each open-headed (relation) arrow labelled f from box A to box B, we introduce variables A, B and f, and assert

$$f : A \to B$$

A constraint is obtained from each multiplicity marking. If f has a marking of ! on its head for example, we generate

$$\textit{all } a\text{: } A \mid \textit{one } a.f$$

For each closed-headed (subset) arrow from box A to box B, we assert

$$A \textit{ in } B$$

and whenever two boxes A and B share such an arrowhead, we add the disjointness constraint

$$\textit{no } A \ \&\ B$$

1.4.2 Qualified Sets

Qualifications cause definitions to be generated. For each box labelled S/e, where S is the set name and e is some expression, we generate a fresh name, S_i say, with the constraint

$$S_i = S \ \&\ e$$

Additionally, each box that appears as a subset (direct or indirect) of S/e is given a fresh name and its own definition in the same way. For example, the diagram of Section 1.2.2 may give

$$
\begin{aligned}
&List_1 = List \ \& \ ZipCityMap.zcl \\
&EmptyList_1 = EmptyList \ \& \ ZipCityMap.zcl \\
&NonEmptyList_1 = NonEmptyList \ \& \ ZipCityMap.zcl
\end{aligned}
$$

1.4.3 Interpreting Concrete Object Models

The formula obtained by conjoining all the constraints obtained from field arrows, classification arrows and defined subsets is then interpreted as follows. Each set variable denotes the set of all objects in the heap that belong to the class or interface with that name; each relational variable denotes the field of that name on the relevant class. A given object in the heap is considered to be a member of the set S if its type is S, or a type that *implements* or *extends* S (in Java terminology). The pair (x, y) is considered to be a member of a relation f when the f field of the object x holds a reference to object y. The heap as a whole satisfies the object model when the formula obtained from the model is true when interpreted in this fashion.

1.4.4 Combining Object Models

Note that there are no implicit constraints that assert disjointness of the top-level sets. We cannot infer, for example, from the object models of Section 1.2 that the sets *String* and *ZipCityMap* are disjoint. This constraint would instead be obtained from the object model that captures the standard class hierarchy (and need not be drawn), which includes these two sets as disjoint subsets of *Object*. Each object model is an invariant in its own right; the meaning of a collection of object models is simply the conjunction of their corresponding formulas.

1.4.5 Abstract Fields

Abstract fields cannot be directly interpreted. Instead, each abstract field must be defined in terms of concrete fields (as illustrated below in Section 1.5.3). The formula obtained from the object model is conjoined with these definitions to give a composite formula with which to evaluate a heap.

1.4.6 Indexed Fields

Indexed fields are encoded as binary relations. Suppose we have an arrow with label $f[I]$ that connects a box labelled A to a box labelled B. We introduce a set variable $Edge_f$ whose elements correspond to edges in snapshots that are instances of the field f, and relations

$f : I \to Edge_f$
$\pi_a : Edge_f \to A$
$\pi_b : Edge_f \to B$

We constrain the projections π_a and π_b to be total functions

all e: Edge$_f$ | *one e.*π_a && *one e.*π_b

The field corresponding to an index i, written $f[i]$, is now defined by

$f[i] = \{v, w \mid (some\ e\colon i.f \mid v = e.\pi_a\ \&\&\ w = e.\pi_b)\}$

This definition allows i to be an arbitrary (set-valued) expression, in which case $f[i]$ is the union of the individual relations.

Domain and range constraints are translated as if for a simple field, but quantifying over all values of the index:

all i $\mid f[i] : A \to B$

Multiplicity markings induce similarly quantified constraints. For example, if the head of the arrow carries an exclamation point, we would generate

all i | *all v*: *A* | *one a.f*$[i]$

In terms of a concrete heap, we interpret the field $f[i]$ on an array object a as the ith element of a. All other indexed fields are abstract and must be defined explicitly.

1.5 Textual Annotations

An object model diagram has limited expressiveness. Constraints that involve objects that are not directly related must usually be expressed textually. Three kinds of constraint are most common: representation invariants, which describe the allowed configurations of objects that form the representation of an abstract type; global invariants, which describe how objects of different abstract types may be configured with respect to one another; and abstract field definitions, which define abstract fields in terms of concrete ones.

1.5.1 Representation Invariants

The representation of *ZipCityMap* as a linked list of *ZipCity* records (Section 1.2.2) would likely be required to satisfy the following invariants:

// no cycles in the list
all m: *ZipCityMap* | *no p*: *m.zcl.*next* | *p in p.+next*

// every list cell has a non-null val field
all m: *ZipCityMap* | *all p*: *m.zcl.*next* & *NonEmptyList* | *some p.val*

// each ZipCity record appears at most once in the list
all m: *ZipCityMap* |
    all zc: *m.zcl.*next.val* | *sole p*: *m.zcl.*next* | *p.val* = *zc*

// each zipcode appears in at most one ZipCity record in the list
all m: *ZipCityMap* |
    all z: *m.zcl.*next.val.zip* | *sole zc*: *m.zcl.*next.val* | *zc.zip* = *z*

Note how the dot operator allows sets of objects to be defined by navigations through the object graph. An expression such as *m.zcl.*next*, for example, which denotes the list cells of the map m, can be read 'start at m, then follow the *zcl* field once, and the *next* field zero or more times'.

The form of these constraints identifies them as representation invariants. Their scope is a single instance of the class *ZipCityMap*, and objects reachable from that instance, evidenced by the omission of references to other sets, and the fact that the transpose operator does not appear. These conditions are sufficient but not necessary; the third constraint

all m: *ZipCityMap* |
    all zc: *m.zcl.*next.val* | *sole p*: *m.zcl.*next* | *p.val* = *zc*

might have been written equivalently

all m: *ZipCityMap* | *all zc*: *ZipCity* | *sole zc.~val* & *m.zcl.*next*

for example.

(In fact, these structural conditions are only part of the story. Whether a constraint is a representation invariant depends on whether the implementations of the methods of the abstract type, in this case *ZipCityMap*, rely on it. The objects whose fields are referred to in the representation invariant will generally also be those that are elided when concrete fields are replaced by abstract fields. But the drawing of a boundary around the representation of a type is fraught with difficulties. It is not clear, for example, whether the String objects that hold the city name and zipcode should be regarded as part of the *ZipCityMap* abstraction. If *ZipCityMap* interns strings, they must be; if a client uses reference equality to compare strings passed into and out of the abstraction, they must not be.)

1.5.2 Global Invariants

Global invariants reflect decisions about the program as a whole rather than particular abstract types. Some examples for a program that uses the list representation of *ZipCityMap*:

// list cells are not shared amongst ZipCityMap objects
all p: List | sole p.~zcl

// ZipCity records are not shared amongst lists
all zc: ZipCity | sole zc.~val

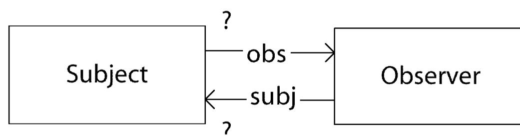
// no String is both a zipcode and a city name
no (ZipCity.zip & ZipCity.city)

and perhaps

// there is only one ZipCityMap
one ZipCityMap

Although sharing constraints are not representation invariants, their violation can cause *representation exposure*, and prevent a representation invariant from being established locally. They can often be enforced by the code of an abstract data type, using a combination of access control mechanisms, strong typing and programmer discipline. To ensure that *ZipCity* records are not shared, for example, the *ZipCityMap* abstraction would likely provide no methods with arguments or results of type *ZipCity* or *List*, and would declare the *zcl* field to be private.

Global invariants are becoming increasingly important due to the popularity of a style of design encouraged by the design patterns literature [9] in which objects are tightly coupled to one another, often linked with mutual references, and interacting by elaborate protocols. The *Observer* pattern, for example, has the following object model:



(assuming the common restriction that an observer may observer at most one subject) and depends on global invariants such as

// each observer points to a subject for which it is an observer
all o: Observer | o in o.subj.obs

// no observer observes itself, directly or indirectly
no o: Observer | o in o.+obs

Note the field *subj* emanating from *Observer*; this is legitimate even if *Observer* is an interface since the object model describes abstract heap structure and is not constrained by the syntactic rules of Java (which do not allow fields to be declared in interfaces). This is a fundamental difference between an object model and a class diagram, which is instead a graphical representation of the program text.

1.5.3 Abstract Field Definitions

Abstract fields can be defined in terms of concrete fields. The view of *ZipCityMap* as a set of records (in section 1.2.4), for example, can be obtained from the list and array representations:

> *// define abstract field zcs in terms of list structure*
> *all m: ZipCityMap | m.zcs = m.zcl.*next.val*
>
> *// define abstract field zcs in terms of array structure*
> *all m: ZipCityMap | m.zcs = m.zca[Index]*

Likewise, the view of *ZipCityMap* as a mapping from zipcodes to cities is given by

> *// define abstract indexed field zipToCity in terms of list structure*
> *all m: ZipCityMap | all z: String | m.zipToCity[z] =*
> *{c: String | some zc: m.zcl.*next.val | zc.zip = z && zc.city = c}*
>
> *// define abstract indexed field zipToCity in terms of array structure*
> *all m: ZipCityMap | all z: String | m.zipToCity[z] =*
> *{c: String | some zc: m.zca[Index] | zc.zip = z && zc.city = c}*

1.6 Discussion

In our discussion below, we compare our approach to object model semantics to previous approaches (1.6.1); compare our annotation language to others (1.6.2); summarize the differences between the interpretations of object models of problems and object models of code (1.6.3); point out some advantages of object models over more conventional specifications (1.6.4, 1.6.5); and finally comment on some deficiencies of our approach (1.6.6, 1.6.7, 1.6.8).

1.6.1 Related Approaches

The most common approach to giving meaning to object models has been translation to a existing formal specification language.

Algebraic languages, especially the Larch Shared Language [10], are the most popular. Such translations [2, 12] benefit from the purity of algebraic specifications, but tend to be rather elaborate, because algebraic operators are total functions, whereas object models are based on relations. Partiality is encoded with special error values; relations are treated differently from functions; navigation expressions call for explicit lifting of functions or relations from scalar to set application; and classification requires the explicit construction of a simulation.

Z [28] is a more natural target, because it is based, like object models, on sets and relations. Rather than using the basic notions of sets and relations,

translations into Z exploit its schema calculus. Some encode classification with schema extension, so that the subbox is mapped to a schema containing the components of the superbox and some additional ones [21]; others use an encoding more like an implementation of subclassing, in which the subbox schema has a component whose type is the schema of the superbox [27]. Like the algebraic approach, both of these assign different types to subboxes and superboxes. Consequently, given a relation r from a box A with a subbox B, and objects a and b of those boxes, the meaning of the expressions $a.r$ and $b.r$ must be obtain differently, since in the latter case some kind of conversion must first be applied. It seems better to treat subbox and superbox objects as belonging to the same type [8].

The Object Constraint Language (OCL) [30] has been used as a target also to give meaning to a subset of its own constraints [18]. OCL is much more complicated than Z or Larch, so the semantics is more of an exercise to demonstrate OCL's expressiveness than to explain its meaning.

Richters and Gogolla give a semantics for OCL [22] that addresses several of its complexities, such as n-ary associations and the operational *iterate* construct. To resolve problems with OCL's flattening rules, they chose to interpret a one-to-many association as a function from an object to a set of objects, and to have navigation result in nested collections that are then implicitly flattened. This introduces a higher-order aspect into the language, which we have been reluctant to do.

1.6.2 Related Languages

The object modelling language of choice for many developers will be the Unified Modeling Language (UML) [25]. UML's constraint language, OCL [30], is more complicated than languages such as Z and Alloy. Navigation expressions are less uniform than ours, and the type system forces one to apply explicit coercions when following relations on a subbox. For example, consider writing a textual constraint saying that, for any *ZipCityMap* object m, the first cell of the list *m.zcl* (if any) holds a *ZipCity* object:

ZipCityMap.zcl.val in ZipCity

In OCL, we would have to write something like

<u>ZipCityMap</u>
zcl.oclIsKindOf(NonEmptyList) implies
zcl.oclAsType(NonEmptyList).val.oclIsKindOf(ZipCity)

because applying the *val* relation to an *EmptyList* is a type error (where in our approach it would just give the empty set). The assertion that all reachable cells hold *ZipCity* objects

*ZipCityMap.zcl.*next.val in ZipCity*

cannot be expressed declaratively at all in OCL, because there is no transitive closure. Instead, one must write an operational definition of a navigation, and then invoke it.

Z can be used, of course, not only to give meaning to object models, but as an object modelling language in its own right. But because Alloy has been tailored to object models, it is a bit more concise and direct than Z in this application, primarily in allowing sets to appear in declarations, and in the syntax of navigation expressions.

Hoare and He [15] give a theory of object models in which each object is characterized by the set of traces of field labels that reach it. In our formalism, objects have identities, so two heaps that have identical structure—and are thus observationally equivalent—may be distinguished. Their theory, in contrast, is fully abstract. Their constraint notation is based on regular expressions, and thus bears a striking similarity to our notation. The trace view does not permit arbitrary relations, however, since, in a snapshot, an object may only have one outgoing field edge with a given label. Representing abstract fields may therefore be a problem.

1.6.3 Code vs. Problem Object Models

Previous work on giving meaning to object models [2, 8, 11, 18, 22], including our own [16], has focused on object models that are used to describe problems or abstract systems rather than heap invariants. This paper was motivated by our discovery that object models that describe implementations require a slightly different interpretation.

- In a problem object model, one typically assumes that all top-level sets or *domains* are disjoint. The presence of interfaces and the root class *Object* makes this assumption infeasible. Instead of using a typed relational logic in which each domain has a basic type associated with it, we treated each class as a subset of the set *Object*; disjointness amongst classes was provided by conjoining the interpretation of the standard class hierarchy.

- In a problem object model, a set may appear at most once in a diagram. We introduced the notion of qualified sets to allow us to instantiate polymorphic containers.

- In a problem object model, all fields are in a sense abstract. It is common to include redundant relations that are defined in terms of other relations and sets, but unlike abstract fields, these do not replace existing relations.

1.6.4 Abstract Fields vs. Abstract Objects

Abstract fields, also called 'specification fields' by some authors, offer several advantages over an approach in which an abstraction function is defined that maps concrete objects to abstract mathematical objects [14, 20].

- Many abstract types have a tuple structure anyway, so it is cleaner to define a collection of fields than a function that maps the concrete object to a tuple, and then a collection of projection functions to deconstruct it.
- Abstract fields can be used to define more fine-grained frame conditions than abstract objects; the JML specification language [19] and the annotation language for the Extended Static Checker [6], for example, both allow *modifies* clauses to indicate which abstract fields of an object may change.
- The abstract object approach still requires explicit mention of the abstraction function in specifications. To specify insertion of an element e into a set s, for example, one has to write something like

  $$s_{post} = s \cup \{e\}$$

  where the *post* suffix indicates dereferencing and application of the abstraction function in the post-state, and other names are implicitly dereferenced and abstracted in the pre-state. Unfortunately, this is not correct: the second mention of s must indeed be dereferenced and abstracted thus, but the mention of e must be treated differently. The abstract mathematical set object contains *references* to objects, so e must not be dereferenced. In the abstract field approach, the set insertion would be specified like this:

  $$s.elems' = s.elems + e$$

  which makes clear the dereferencing of s but not e.

The abstract object approach seems to work well in a traditional data abstraction setting in which objects are immutable and can be viewed in terms of pure mathematical values, but is less useful in object-oriented programs where mutation and sharing are common.

1.6.5 Fields as Relations

In an object model, every field is treated as a relation. In contrast, more expressive specification approaches allow fields to map object references to arbitrary mathematical objects. JML, for example, allows fields to take on values defined by an algebraic specification. An abstract field may map an object to a priority queue, for example. In the object model approach,

one would either have to make do with a partial specification (ignoring the ordering of objects in the queue entirely, for example), or model the queue using indexed fields, or perhaps even by introducing objects purely for specification purposes, in the style of model-oriented languages.

The object model approach has a few important advantages. It is simpler, is closer to the true structure of the heap, and more easily represented graphically. Navigation expressions allow reachability relationships to be described uniformly and simply; with arbitrary types as the values of fields, images of sets can no longer be taken in the obvious way, and fields cannot be navigated backwards. Object models also have a strong similarity to the kinds of shape graph computed by static analyses, suggesting the possibility of checking conformance of code to object models fully automatically.

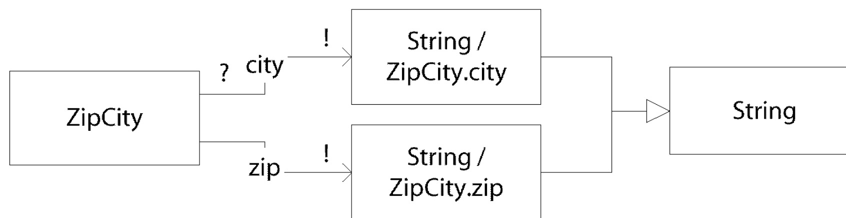
1.6.6 Qualified Sets

The ability to split sets into subsets, and use the same label for two different boxes in the object model is crucial. It allows representation details to be shown, since one can distinguish an array used in the representation of one class from an array used in another, for example. Without splitting, no generic structure can be instantiated. A program that used the *Observer* pattern, for example, would not be able to distinguish observers of different subject classes.

Splitting might also be useful for primitive classes. For example, if we represented the images of the *zipcode* and *city* fields as defined subsets of *String*, we would draw an object model that incorporates graphically the constraint

// no String is both a zipcode and a city name
no (ZipCity.zip & ZipCity.city)

of section 1.5.2:



We considered viewing splitting as instantiation of type parameters: regarding two boxes labelled *Vector* as representing *Vector*$[S]$ and *Vector*$[T]$ for distinct types S and T, for example. But this does not sit well with Java's lack of parametric polymorphism, and is more complicated than the

scheme based on defined subsets presented here. One can always qualify a set according to its contents: *Vector / S.~elts*, for example, for *Vector*[*S*].

Our scheme is an instance of a more general scheme developed by Rinard and Kuncak [23], in which arbitrary sets can be defined over the actual heap state. They allow a set representing a class to be partitioned, for example, into subsets according to the value of a particular field. Their aim is to bridge the gap between shape graphs that can be extracted by static analysis and abstract object models that capture essential design constraints.

A deficiency of our approach is that there is no easy way to indicate that the subset used in one context is not used elsewhere. One cannot say, for example, that the set of *List* objects used in the representation of *ZipCityMap* is disjoint from other sets of *List* objects; such constraints would have to be added explicitly for each other such set (eg, by including both qualified *List* sets as disjoint subsets of *List* in a single diagram).

1.6.7 Indexed Fields

The encoding of indexed fields described in Section 1.4.6 is clumsy. Alloy in fact offers indexed relations as a built-in feature, and they are supported by our analysis tool. But indexed relations are also problematic, and in a future version of the language we expect to support relations of arbitrary arity: 1 for sets and scalars, 2 for binary relations, 3 for relations with one index, and so on. This will allow the dot operator to be used uniformly, so that $e.f[i]$ will be equivalent to $e.(i.f)$. A similar view is taken by Richters and Gogolla in their semantics for OCL, in which fields are modelled as hyperarcs [22].

We have been a bit cavalier in introducing index sets, such as *Index* in Section 1.2.3. An index set may correspond to a class of objects, or, as in the case of *Index*, it may be a specification artifact (in the style of model-based specification).

1.6.8 Extensional Equality

In some invariants, we want an extensional notion of equality that equates two (possibly distinct) objects when their fields have the same values. In Java, for example, two distinct strings may contain the same sequence of characters. Instead of the representation invariant of section 1.5.1

```
// each zipcode appears in at most one ZipCity record in the list
all m:ZipCityMap |
        all z: m.zcl.*next.val.zip | sole zc: m.zcl.*next.val | zc.zip = z
```

we might want to say that no two *ZipCity* records in the list contain strings with the same sequence of characters. One might define an equivalence

relation *eq* as an abstract field, so that *x in y.eq* is true when *x* and *y* are equivalent. The invariant would then become

all m: *ZipCityMap* | *all z*: *m.zcl.*next.val.zip* |
sole zc: *m.zcl.*next.val* | *z in zc.zip.eq*

Acknowledgments

This work grew out of an attempt to use object models in an undergraduate software engineering course at MIT, as a pervasive framework for explaining the Java heap model, rep invariants, abstraction functions, and global invariants. It has benefited from the criticisms of my co-lecturer, Michael Ernst, and of our teaching assistants, especially Felix Klock and Allison Waingold. I am very grateful also to Alan Fekete, Michael Jackson and Annabelle McIver who read an early draft, spotted mistakes and suggested many improvements, and to Ed Wang who converted the paper from FrameMaker to Latex.

Martin Rinard and Viktor Kuncak contributed essential ideas. Martin had developed the idea of 'content-based classification', in which objects would be classified according to the values of their fields. Viktor suggested that an object model be viewed in the context of the class hierarchy, so that the standard relationships between a class and *Object* can be omitted.

This research was funded by an Information Technology Research grant (#0086154) from the National Science Foundation, by a grant from NASA, and by an endowment from Douglas and Pat Ross.

1.7 References

[1] Serge Abiteboul and Richard Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*,Vol. 12, No. 4, December 1987, pp. 525–565.

[2] Robert H. Bourdeau and Betty H.C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, October 1995.

[3] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

[4] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, 1994.

[5] Peter P. Chen. The Entity-Relationship Model-Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, (1976), pp. 9–36.

[6] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *Extended Static Checking*. Research Report 159, Compaq Systems Research Center, December, 1998.

[7] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1998.

[8] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Malcolm Shroff. Exploring the Semantics of UML Type Structures with Z. *Proceedings of the Formal Methods for Open Object-based Distributed Systems* (FMOODS'97), Canterbury, England, July 1997. Chapman & Hall, London, 1997.

[9] Erich Gamma, Richard Helm,Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, October 1994.

[10] John V. Guttag, James J. Horning, and Andres Modet. *Report on the Larch Shared Language: Version 2.3.* Technical Report 58, Compaq Systems Research Center, Palo Alto, CA, 1990.

[11] Ali Hamie, John Howse and Stuart Kent. Interpreting the Object Constraint Language. *Proceedings of Asia Pacific Conference in Software Engineering*, IEEE Press, 1998.

[12] Ali Hamie, John Howse and Stuart Kent. Navigation expressions in object-oriented modelling. *Proceedings Fundamental Approaches to Software Engineering* (FASE'98), *European Joint Conferences on the Theory and Practice of Software* (ETAPS'98), Lisbon, Portugal, March 1998. Springer-Verlag, LNCS 1382, 1998.

[13] Michael Hammer and Dennis McLeod. Database description with SDM: a semantic database model. *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981, pp. 351–386.

[14] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), pp. 271–281, 1972.

[15] C.A.R. Hoare and Jifeng He. A trace model for pointers and objects. *Proc. 13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 14-18, 1999. Springer-Verlag, LNCS 1628, 1999. Reprinted in this volume.

[16] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation.* Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000. Latest version available at: http://sdg.lcs.mit.edu/∼dnj/publications. To appear, *ACM Transactions on Software Engineering.*

[17] Daniel Jackson. Automating first-order relational logic. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*. San Diego, November 2000.

[18] Stuart Kent, Stephen Gaito, Niall Ross. A meta-model semantics for structural constraints in UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral specifications for businesses and systems*, chapter 9, pages 123–141. Kluwer Academic Publishers, Norwell, MA, September 1999.

[19] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Preliminary design of JML: a behavioral interface specification language for Java*. Department of Computer Science, Iowa State University, TR #98-06j, June 1998, revised May 2000. Available at http://www.cs.iastate.edu/∼leavens.

[20] Barbara Liskov with John Guttag. *Program Development in Java*. Addison-Wesley, 2001.

[21] E.-R. Olderog and A.P. Ravn. Documenting design refinement. *Proc. ACM SIGSOFT Workshop on Formal Methods in Software Practice*, Portland, Oregon, August 2000, pp. 89–100.

[22] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL, *Proc. 17th Int. Conf. Conceptual Modeling* (ER'98), 1998, Springer-Verlag, LNCS 1507, pp. 449–464.

[23] Martin Rinard and Viktor Kuncak. *Object Models, Heaps and Interpretations.* Technical Report 816, MIT Laboratory for Computer Science, Cambridge, Massachusetts, January 2001.

[24] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

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

[26] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981, pp. 140–173.

[27] Malcolm Shroff and Robert B. France. Towards a formalization of UML class structures in Z. *Proceedings of Twenty-First Annual International Computer Software and Applications Conference* (COMPSAC'97), pages 646–651. IEEE Computer Society, 1997.

[28] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second edition, Prentice Hall, 1992.

[29] Toby J. Teorey, DongQing Yang and James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2), June 1986, pp. 197–222.

[30] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.