Alloy: A New Object Modelling Notation

Daniel Jackson
MIT Laboratory for Computer Science
dnj@lcs.mit.edu

Abstract

Alloy is a lightweight, precise and tractable notation for object modelling. It attempts to combine the practicality of UML's static structure notation with the rigour of Z, and to be expressive enough for most object modelling problems while remaining amenable to automatic analysis.

Alloy has a textual notation, of which a subset is also expressible graphically. It has a simple set-based semantics, and a type system that, by treating scalars as singleton sets, allows relations and functions to be treated uniformly, and sidesteps the problem of undefined expressions.

To answer the obvious question—why yet another language?—the paper gives Alloy, Z and UML versions of a small example and compares them in detail.

Keywords

Object modelling, formal specification, static structure diagram, Z specification language, Unified Modeling Language, Object Constraint Language, relational calculus.

1 Introduction

An object model describes a state space, in which each state is viewed abstractly as a collection of objects and their inter-relationships. Object models are used primarily in the design of object-oriented programs. In the specification phase, the objects represent conceptual entities, often drawn from the problem domain. In the design phase, the objects correspond to members of classes, and the model as a whole may be viewed as an abstraction of the set of reachable heap states. Object models can also be useful in other applications that involve relational structure, eg. for documenting the constraints of an architectural style or integration framework.

Alloy is a new object modelling notation that was designed to meet three criteria: first, to be lightweight—small and easy to use, and capable of expressing common properties tersely and naturally; second, to be precise—that is, having a simple and uniform mathematical semantics; and, third, to be tractable—amenable to efficient and fully automatic semantic analysis.

I designed Alloy because no existing language satisfied all three criteria. UML [Rat97], whose object modelling facility consists of a graphical notation (the static structure diagram) and OCL [WK99], a textual constraint language, is not precise, and perhaps not lightweight either, at least by comparison to its much simpler predecessor OMT [Rum91]. Z [Spi92] is precise but intractable, and not a natural fit for object modelling. NP [JD96a], the language of my Nitpick checker [JD96b], had similar design goals to Alloy, but is also not well suited to object modelling, with its Z-style declarations and a lack of quantifiers.

Alloy takes features from each of these. From UML, it takes diagrams and the navigation syntax for relational image, whose origins are in OMT and Syntropy [CD94] respectively. From Z, it takes set-based semantics, in which classes of objects are modelled as sets of elements drawn from primitive types. From NP, it takes the idea of structuring the model to expose opportunities for checking.

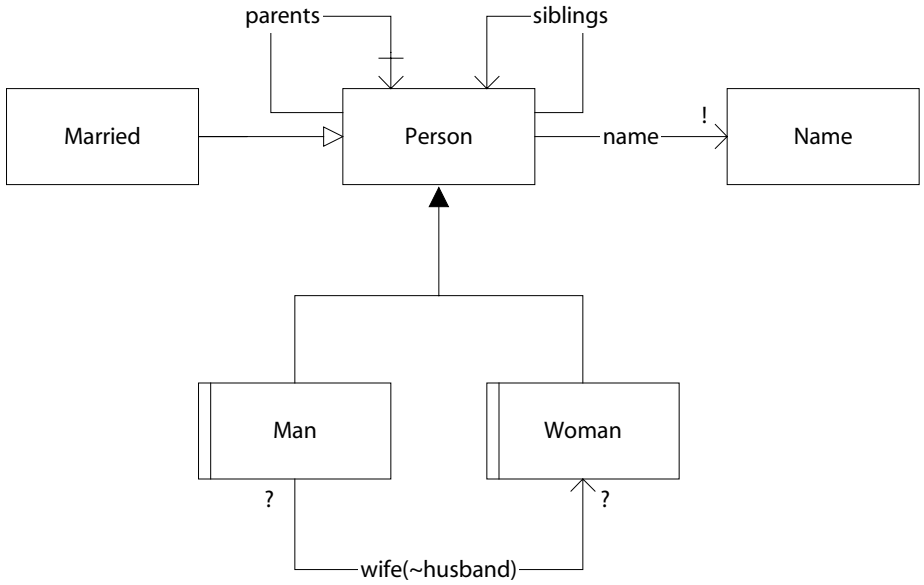
The challenge of Alloy's design has been to combine these features smoothly. The key idea is a simple type system that associates primitive types implicitly with sets that are not declared as subsets of other sets, and that treats scalars as singleton sets. This allows a diagram to be given a direct textual counterpart (with the additional benefit that a model can always be expressed in text alone), and makes navigation expressions uniform, where in Z, functions and relations are syntactically distinct. It also sidesteps the partial function problem.

My work also adds to the long series of papers that use ideas from languages such as Z to clarify the semantics of informal notations such as UML. This paper differs, however, in two respects. First, it argues that the insights of formal specification can be used to simplify UML, and to eliminate troublesome features, where previous work has tended to use formalization to lend support to its complexities. Second, the paper addresses mutability in detail, a vital aspect of object modelling that is usually overlooked. UML provides a variety of markings for association ends that include the keyword "frozen". In Alloy, this is promoted to a key language feature—the notion of static sets and relations—and given a simple and uniform semantics.

Alloy was designed hand-in-hand with its checker, Fox (Fast Object Constraint Solver), to be described in a forthcoming paper. Several features of the language were either motivated by the possibility of performing checks—for example, the distinction between invariant, condition and definition schemas—or were added only when a checking mechanism had been devised. Navigation expressions, for example, depend on quantifiers; these could not be handled by previous methods [Jac96b, Jac98] but are handled efficiently by a new method in Fox.

2 An Example

To see how Alloy works, consider a model describing family members and their interrelationships (Figure 1). The diagram is equivalent to the declarations of the textual model: that is, to the schemas marked *domain* and *state*.



```
model Family {
domain {Person, Name}
state {
  partition Man, Woman : static Person
  Married : Person
  parents : Person -> static Person
  siblings : Person -> Person
  wife (~husband) : Man ? -> Woman ?
  name : Person -> Name !
  }
def siblings {
  all a, b | a in b.siblings <-> (a.parents = b.parents)
  }
 inv Basics {
  all p | some p.wife <-> p in Man & Married
  no p | p.wife / in p.siblings
  all p | (sole p.parents & Man) && (sole p.parents & Woman)
  no p | p in p.+parents
  all p, q | p.name = q.name -> no (p.parents & q.parents)
  }
op Marry (m: Man!, w: Woman!) {
  m not in Married && w not in Married
  m.wife' = w
  all p: Man - m | p.wife' = p.wife
  all p | p.name' = p.name
  all p | p.parents' = p.parents
  Person' = Person
  }
 assert HusbandsWife {
  all p : Married & Woman | p.husband.wife = p
  }
}
```

Figure 1: *Alloy model for a family tree*

Sets, Domains, Types and Relations

Each box in the diagram denotes a set of objects. An object is an abstract, atomic and unchanging entity; the state of the model resides in the relations between objects, and in the membership of objects in sets. There are two kinds of arrow connecting boxes. An arrow with a closed head denotes subset: ***Man***, ***Woman*** and ***Married*** are subsets of ***Person***. Subsets that share an arrow are disjoint; if the arrow head is filled, the subsets are exclusive. So ***Man*** and ***Woman*** partition ***Person***: every ***Person*** is either a ***Man*** or a ***Woman***. An arrow with an open head denotes a relation: *name*, for example, is a relation that maps ***Person*** to ***Name***.

Sets without supersets, such as ***Name*** and ***Person***, are called *domains*, and are implicitly disjoint. A type ***ty*** (***D)*** is implicitly associated with each domain ***D***. The expression (***Man*** + ***Woman***), denoting the union of the two sets ***Man*** and ***Woman***, is legal because both ***Man***

and *Woman* have the type *ty* (*Person*); but the expression (*Man* + *Name*) is not, because *Man* has the type *ty* (*Person*) and *Name* has the type *ty* (*Name*).

Markings at the ends of relation arrows denote multiplicity constraints: ! for exactly one, ? for zero or one, * for zero or more and + for one or more. Omission of a marking is equivalent to *. So *name*, for example, maps each *Person* to exactly one *Name* (by the mark at the *Name* end), and maps zero or more members of *Person* to each *Name* (by the mark at the *Person* end). The composite label *wife*(~*husband*) declares a relation and its transpose: *wife* maps *Man* to *Woman*, *husband* maps *Woman* to *Man*, and whenever one maps object p to object q, the other maps q to p.

Mutability Constraints

Alloy includes some basic temporal constraints. Although very limited in expressiveness, these turn out to be invaluable in practice—and also more subtle than they might at first appear. The stripes down the sides of the boxes labelled *Man* and *Woman* say that those sets are *static*. The members of a static set may not migrate to other sets. So a *Person* may not at one point in time be a *Man* and at another point not be a *Man*, but since the set *Married* is not static, a *Person* may be *Married* at one point and not *Married* at another. Domains are implicitly static, so a *Person* may not become a *Name*.

The hatch mark behind the arrowhead of the parents relation declares the relation to be *right-static*. For any *Person*, the set of objects it is mapped to by the *parents* relation is fixed during its lifetime: in other words, a *Person* may not change parents. This relation is not left-static, however, a parent may give birth to more children. The model thus expresses a 'family tree' view of existence: birth causes an addition to the state, but death does not cause a deletion.

These temporal constraints have fundamental consequences for implementation. If *Person* were implemented as a class with a field representing the *parents* relation, the static property of the relation indicates that the set of objects pointed to by the field is fixed and that an immutable datatype would be appropriate. In Java, for example, we could then use an array rather than a Vector. Similarly, the fact that *Man* and *Woman* are static sets would allow us to represent them as subclasses of *Person*; *Married* on the other hand, must be represented as object state.

Syntropy's 'state types' [CD94] corresponds to non-static sets. I have chosen to mark sets when static, rather than vice versa, to obey the principle of monotonicity: that every addition to a diagram should strengthen the description, and never weaken it. This also motivated the choice of UML- rather than OMT-style multiplicity markings. In OMT, a relation is one-to-one if unmarked; adding solid blobs eliminates this constraint.

Textual Declarations

The domain and state schemas of the textual version correspond exactly to the diagram. The domain schema lists the sets that are to be regarded as domains, in this case *Person* and *Name*. The state schema declares the remaining sets, and the relations. A declaration of the form

```
S:T
```

declares *S* to be a subset of the set *T*; a declaration of the form

```
r:S->T
```

declares *r* to be a relation from *S* to *T*. The types are implicit; since *Man*, for example, is declared to be a subset of the domain *Person*, which has implicit type *Person*, we can infer that the left-hand type of the *wife* relation is *Person*. Multiplicity is indicated with the same markings used in the diagram; mutability constraints are indicated with the keyword *static*.

Organization of Constraints

Only the most basic constraints can be expressed graphically. The remaining constraints are expressed textually. These are divided according to their function. Definitions, namely constraints that establish the values of extra components introduced for convenience, appear in schemas marked by the keyword *def*; here, there is just one, for the relation *siblings*. Invariants, namely indicative properties of the domain, appear in schemas marked by the keyword *inv*; in this case, there is only one, and it has the name *Basics*. Assertions are constraints that are expected to follow from the invariants and definitions, and appear in schemas marked by the keyword *assert*. Assertions about operations are also possible, but none is shown here.

Navigation and Quantification

The definition of *siblings* consists of a single constraint: that, for all *a* and *b* that are members of *Person*, *a* is a sibling of *b* if *a* and *b* have the same parents. The expression *b.siblings* is called a *navigation*: intuitively, evaluation starts at some *b*, and the *siblings* relation is 'navigated' to the set of members of *Person* that the relation maps *b* to. No quantifier bounds need be declared; they are automatically inferred. In this case, since parents is a relation whose left type is *Person*, *a* and *b* are inferred to have type *ty* (*Person*).

Expressions

All expressions in Alloy denote sets of objects. Expressions are formed with the conventional set operators, and with navigation expressions. The expression (*Man & Married*), for example, denotes the intersection of the sets *Man* and *Married*. The result of a navigation is always a set; functions are treated as relations with a special property, so the expression *p.wife* either evaluates to the empty set (when *p* is not mapped by *wife*) or to a singleton set (containing the *wife* of *p*). The term *some e* is true when the expression *e* is not the empty set. So the first constraint in *Basics* says that every person who has a *wife* is a married man.

Even variables take on set values, although these are implicitly constrained always to be singleton sets. The term *e1 in e2* is true when the set denoted by *e1* is a subset of that denoted by *e2*, so the term (*p in Man & Married*) has the same meaning it would have were *p* to denote an element and *in* to denote set membership. Encoding scalars as sets is a useful trick that allows functions and relations to be treated uniformly (since there is no function application distinct from relational image), and resolves the problem of partial functions applied outside their domain in a simple and pragmatic fashion. If *f* is a function that does not map *x*, the expression *x.f* denotes the empty set, so if *y* is a scalar (and thus a singleton set), *x.f = y* is false.

It is often convenient to state, as a side-condition, that a navigation expression yields a non-empty set. The second constraint of *Basics*, for example, says that there is no person whose *wife* is also a sibling. For a person *p* with no *wife*, the expression *p.wife* would evaluate to the empty set, and the term *p.wife in p.siblings* would be true. The term

```
e1 /in e2
```

is like *e1 in e2* but adds the condition that *e1* is non-empty.

Suffixes and Relational Operators

There are no operators for combining relations; all expressions denote sets. However, in a navigation expression, the transpose or closure of a relation may be used. In the constraint

```
no p | p in p.+parents
```

for example, *p.+parents* is the image of *p* under the transitive closure of the *parents* relation—the set of objects obtained by following parents once, then again, and again, and so

on until no further objects are added—in other words, *p*'s ancestors. The constraint thus states that no person is his or her own ancestor.

3 Comparison to Z

Z is a "system of notation for building structured mathematical theories", and the language itself has "no necessary connection with computer programming" [Spi92, p.128]. This explains both the strength and the weakness of Z. It is, on the one hand, remarkably powerful, and can be used to model many different aspects of a software system. On the other hand, its very generality prevents it from fitting any problem perfectly (and is a liability for automatic checking).

Here, we'll consider only how Alloy's constructs make object modelling easier and more natural. But there are deeper advantages to tailoring the language to the problem. A more tightly fitting "frame" can paradoxically make problems easier to solve; moreover, Z's reluctance to treat the connection to software as anything more than informal convention results in descriptions whose meaning is hard to pin down [MJa95].

A Z specification for our problem could be written in many different ways. So as not to be accused of comparing with a strawman, I have written two versions that typify extremes of Z usage. The basic declarations (in the schema *FamilyDecls*) are shared, but the constraints appear twice, once in the schema *FamilyInv1*, and again in *FamilyInv2*. In both cases, the constraints match the Alloy constraints of Figure 1, although the third Alloy constraint is given, for readability, as two separate constraints in Z.

FamilyInv1 uses relational operators in place of quantifiers. The formulas that result—once called "Sorensen shorties"—are terse and elegant, but rarely natural. This style has been used effectively in many published Z specifications; see, for example, the Simple Assembler example in [Hay93]. NP supported only this style, and I found that many readers, especially novices and programmers without mathematical background, are uncomfortable with it. The first constraint can also be expressed with relational operators alone, but Z's lack of either a complementation operator or a universal relation constant with implicit type makes it rather unwieldy. *FamilyInv2* is what most novices would produce, and corresponds to the style of Alloy and UML.

$$[PERSON, NAME]$$

FamilyDecls

$$Person, Man, Woman, Married : \mathbb{F}\ PERSON$$
$$Name : \mathbb{F}\ NAME$$
$$parents, siblings : PERSON \leftrightarrow PERSON$$
$$wife, husband : PERSON \rightarrowtail PERSON$$
$$name : PERSON \rightarrow NAME$$

$$\langle Man, Woman \rangle \text{ partition } Person$$
$$Married \subseteq Person$$
$$parents \in Person \leftrightarrow Person$$
$$siblings \in Person \leftrightarrow Person$$
$$wife \in (Man \cap Married \leftrightarrow Woman \cap Married)$$
$$husband = wife\sim$$
$$name \in Person \leftrightarrow Name$$

FamilyInv1

FamilyDecls

$siblings = \{a, b: PERSON \mid parents(\{a\}) = parents(\{b\}) \bullet a \mapsto b\}$

disjoint $\langle wife, siblings \rangle$

$parents \rhd Man \in PERSON \rightarrow\!\!\!\!\!\!+\; PERSON$

$parents \rhd Woman \in PERSON \rightarrow\!\!\!\!\!\!+\; PERSON$

disjoint $\langle parents^+, \mathrm{id}\ PERSON \rangle$

disjoint $\langle name\ ;\ name\sim\ ,\ parents\ ;\ parents\sim \rangle$

FamilyInv2

FamilyDecls

$\forall a, b : Person \bullet a \mapsto b \in siblings \Leftrightarrow parents(\{a\}) = parents(\{b\})$

$\nexists p: Person \bullet p \in \mathrm{dom}\ wife \wedge wife(p) \in siblings(\{p\})$

$\forall p: Person \bullet \#(parents(\{p\}) \cap Man) \leq 1$

$\forall p: Person \bullet \#(parents(\{p\}) \cap Woman) \leq 1$

$\nexists p: Person \bullet p \in parents^+(\{p\})$

$\forall p, q: Person \bullet name(p) = name(q) \Rightarrow parents(\{p\}) \cap parents(\{q\}) = \varnothing$

HusbandsWife

FamilyDecls

$\forall p : Married \cap Woman \bullet$

$p \in \mathrm{dom}\ wife \wedge wife(p) \in \mathrm{dom}\ husband \wedge husband(wife(p)) = p$

Op

Δ *FamilyInv*

$\forall p: Person \cap Person' \bullet$

$p \in Man \Leftrightarrow p \in Man'$

$\wedge\ p \in Woman \Leftrightarrow p \in Woman'$

$\wedge\ parents(\{p\}) = parents'(\{p\})$

Marry

Op

$m?, f?: PERSON$

$m? \in Man \setminus Married \wedge f? \in Woman \setminus Married$

$Person = Person' \wedge parents = parents' \wedge name = name'$

$wife' = wife \oplus \{m? \mapsto f?\}$

Figure 2: Z specification corresponding to Figure 1

Types and Sets

FamilyDecls shows how remarkably cumbersome it is to encode an object model in Z. Given types must be introduced for each of the domains, and explicitly. This is only a minor inconvenience; more serious is that only the given types may be appear on the right-hand side of type declarations, and so the constraints implicit in the Alloy declarations must be given explicitly in the body of the schema. In short, the translation of a UML diagram into Z is untidy, albeit mechanical.

In most Z specifications, this is not a problem. Z specifications, unlike object models, do not generally introduce separate state components for the sets of elements mapped, and mapped to, by relational components. On the contrary, it is common to make frequent use of expressions such as *dom r* to refer to the elements mapped by the relation *r*.

Alloy's inference of the bounds of quantified variables is a useful feature also not present in Z. Of course, Alloy can offer these features only because it is simpler and more constrained than Z. I am not suggesting that, in the context of notions such as schemas as types, it would make sense to infer types.

Temporal Constraints

Z not only has "no necessary connection with computer programming"; it does not even embody the notion of a state machine. States and state transitions are fundamentally no different, and there is no notion of constancy. It is not possible in Z to declare a state component and constrain it to be fixed. A Z specification of a file hierarchy with a fixed root, for example, would either include a template operation that does not change the root (and which would be imported by all operations by convention), or would define *root* as a global constant, violating the modularity of the specification.

Surprisingly, the notion of constancy–that the value of a state component does not change—is not of much use in object modelling. Individual objects are continually created and destroyed; what tends to be constant is the relationship between an existing object and other objects. Of course, we could avoid this problem by modelling relations as fields of objects, as in [Hall90]. But this approach is not in the spirit of object modelling, whose essence is precisely the global representation of state. In this respect, object modelling is certainly not "object-oriented". Alloy's contribution here, with the notion of static sets and relations, is to show how the notion of constancy with respect to individual objects can be expressed while retaining global state, and without assigning state to objects.

Navigation, Sets and Scalars

The constraints of *FamilyInv2* illustrate in particular two inconveniences of Z. First is the need to cast scalars to singleton sets prior to taking the relational image, and the differing syntax for function application and relational image. From an object modelling point of view, it seems odd that changing the multiplicity of a relation should also require a change in the syntax of the expressions in which the relation appears. Alloy avoids this by treating scalars as singleton sets.

Second, Alloy requires no guard (such as the test *p in* dom *wife*) to ensure that expressions involving applications of partial functions are defined. Had *name* been declared as partial rather than total, the final constraint would have been

$$\forall\, p, q: Person \bullet p \in \mathrm{dom}\ name \land q \in \mathrm{dom}\ name \land name(p) = name(q) \Rightarrow parents(\{p\}) \cap parents(\{q\}) = \varnothing$$

but the Alloy constraint would have remained

```
all p, q | p.name = q.name -> no (p.parents & q.parents)
```

Schema Roles

The role of a schema in a Z specification is determined by informal conventions. All schemas that mention no primed variables are methodologically equivalent. No distinctions are made between very different parts of the specification: declarations of state components; definitions of additional, redundant components; state invariants; state conditions used in invariants and operations; and assertions about invariants, conditions or operations.

This means that any such distinctions must be pointed out informally in accompanying text. The assertion of the Alloy model would likely be written as the free-standing theorem

$$\forall \mathit{FamilyInv} . \mathit{HusbandsWife}$$

It also limits the possibilities of tool support. The form of an Alloy model suggests checks to be performed: that assertions are valid; that definitions determine the value of the defined state component; and that invariants are preserved by operations.

Relational Formulas

Relations are not first-class citizens in Alloy. The statement that the operation *Marry* adds a new pair to the *wife* relation—trivial in Z—cannot be written directly in Alloy. One possible remedy to this deficiency, currently being investigated, is to include an assignment statement in the language. The statement

$$m.wife := f$$

would then be treated as a shorthand for two constraints:

$$m.wife' = w$$

and the frame condition

$$all\ p: Person - m \mid p.wife' = p.wife$$

Lexical and Typographic Issues

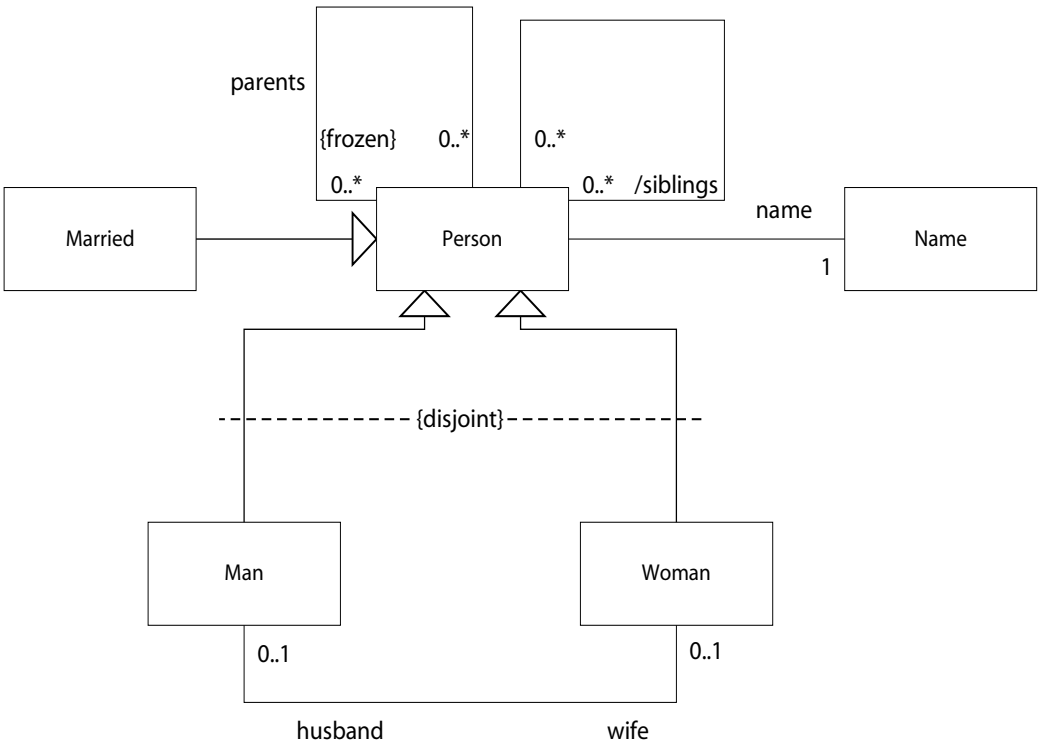
I hesitate to criticize Z for its elegant typography. But I wonder whether it may not, in fact, be a serious impediment to its widespread use. Z's mathematical symbols are not available in standard fonts – not even in Adobe's massive Mathematical Pi – and schema boxes are hard to draw in standard word processors. Neither adopting an amateur font, nor taking on Latex, are palatable options outside academia. For most software engineers, just including Z within a document is hard. Perhaps unicode will eventually solve this problem—Lucida Sans, for example, has a wonderful arrow collection—but in the meantime, an ASCII notation seems to have an advantage.

4 Comparison to UML

The Unified Modeling Language (UML) [Rat97] is a combination of the notations of Rumbaugh, Booch and Jacobson. For object modelling, it provides a graphical "static structure" notation, and OCL (Object Constraint Language) [WK99], a textual notation originally developed at IBM. UML has the backing of a large consortium that includes Microsoft, Oracle, HP and IBM, and was made a standard by the Object Management Group in 1997.

OCL was designed to be less intimidating to practitioners than languages such as Z: it makes no use of Greek letters and many of its notions will be familiar to object-oriented programmers. But conceptually it is far more complicated than Z. It employs the same basic logical and set-theoretic notions of Z and Alloy, but applies these in the context of a programming model that includes subclass and parametric polymorphism, operator overloading, multiple inheritance, and introspection.

These complexities are a formidable obstacle to giving OCL a semantics, and in many cases seem to make the notation harder to use.



Gross Structure

The graphical notation of UML has no textual counterpart, so every model must include a diagram. Moreover, those diagrammatic constraints that are expressible in the textual notation do not map to it straightforwardly. The fact that ***Person*** is partitioned into ***Man*** and ***Woman***, represented in the diagram by marking ***Person*** as abstract and ***Man*** and ***Woman*** as disjoint, would be written

Person
self.oclIsKindOf (Man) implies not self.oclIsKindOf (Woman)
self.oclIsKindOf (Woman) implies not self.oclIsKindOf (Man)
self.allInstances->select (oclIsType = Person)->isEmpty

Some graphical elements are described so informally in the UML documentation that their significance is unclear. Whether an association's property of being an aggregation is expressible in OCL, for example, is hard to tell. UML does not seem to make the distinction between static and non-static sets, although this distinction is present in Catalysis [DW98].

The graphical notation conveniently distinguishes defined components (with a backslash before the name), but OCL does not seem to have a corresponding notion. Nor does UML separate constraints of the model proper from assertions—redundant constraints that are intended to follow.

Contexts, Classes & Types

Each constraint appears in the context of a particular class of objects, and is implicitly universally quantified. The Alloy assertion

```
all p : Married & Woman | p.husband.wife = p
```

for example, appears under ***Woman***. Contexts make constraints more terse, but can induce an arbitrary structure on the model as a whole. This constraint, for example, might equally well have appeared under ***Married***. It also seems odd that an operation such as *Marry*, which is no more about one object than another, should need to be assigned to a class.

In place of the traditional syntax of first-order logic, UML uses a linear form in which formulas are treated as navigation expressions of boolean type. A constraint about all elements of a set is written as an expression denoting the set, suffixed with a parameterized formula that is evaluated over all its members. This form is intuitively appealing, but can be cumbersome for elaborate constraints, or for quantifications over more than one variable. The last constraint of ***Person***, for example, corresponds to the Alloy constraint

```
all p, q | p.name = q.name -> no (p.parents & q.parents)
```

The interpretation of UML expressions is complicated by rules that determine when implicit flattening operators are applied. An expression containing one relation denotes a set; but an expression with two denotes a bag. As a result, the expressions ***e.r*** and ***e.p.q*** are not equivalent when ***r*** is the relation join of ***p*** and ***q***, and one cannot define two relations, ***parents*** and ***grandparents***, so that ***p.parents.parents*** and ***p.grandparents*** have the same meaning.

Classes are not treated semantically as simple sets in UML. One cannot define the men that are married as the intersection of two sets, as in Alloy or Z. Instead, a constraint about such a set must employ built-in operators to type-case and downcast objects. This can

```
Person
self.siblings = Person.allInstances->select(parents = self.parents)
self.parents->select (oclIsKindOf (Man))->size <= 1
self.parents->select (oclIsKindOf (Woman))->size <= 1
not self.parent->includes (self)
Person.allInstances->forall (p, q | p.name = q.name implies
  p.parents->intersection (q.parents)->isEmpty)

Man
self.wife.notEmpty implies self.oclIsKindOf (Married)
self.oclIsKindOf (Married) implies self.wife.notEmpty
self.wife->intersection (self.siblings)->isEmpty

Woman
self. oclIsKindOf (Married)-> self.husband.wife = self

Man :: Marry (w: Woman)
pre: not self.oclIsKindOf (Married)
     not w.oclIsKindOf (Married)
post: self.wife = w
    Man.allInstances->forall (m | m != self implies m.wife@pre = m.wife)
    Man.allInstances->forall (m | m.name@pre = m.name)
    Man.allInstances->forall (m | m.parents@pre = m.parents)
```

Figure 3: UML Model corresponding to Figure 1
(accompanying diagram shown on previous page)

make life difficult. The following constraint from the UML semantics [Rat97, Section 9.3]

Collaboration

```
// if a ClassifierRole or an AssociationRole does not have a name then it should be the only one with a
particular base.

    self.ownedElement->forAll ( p |
            (p.oclIsKindOf (ClassifierRole) implies
                    p.name = '' implies
                            self.ownedElement->forAll ( q |
                                    q.oclIsKindOf(ClassifierRole) implies
                                            (p.oclAsType(ClassifierRole).base =
                                                    q.oclAsType(ClassifierRole).base implies p = q) ) )
and
            (p.oclIsKindOf (AssociationRole) implies
                    p.name = '' implies
                            self.ownedElement->forAll ( q |
                                    q.oclIsKindOf(AssociationRole) implies
                                            (p.oclAsType(AssociationRole).base =
                                                    q.oclAsType(AssociationRole).base implies p = q) )
)
```

for example, would be written in Alloy as

```
all c | all e, f : c.ownedElement | (no e.name) && (e.base = f.base) -> e = f
```

The notion of type and subtype in UML has some subtle consequences that limit expressiveness. Since constraints are implicitly inherited by subclasses, existential quantifiers cannot appear outermost in a formula. For example, the Alloy constraint

```
some p: Person | some p.wife
```

(that there is a person with a ***wife***) cannot be expressed in UML. Were it to be expressible, it would have to be placed in the context of ***Person***, and then automatically inherited by ***Woman***, resulting in the additional constraint that some woman has a ***wife***. This presumably explains the odd rule that multiple iterators are allowed only for universal and not existential quantifiers [WK99, p.47].

Functions

UML, like Z and unlike Alloy, treats functions and relations differently. The result of a navigation through a function is a scalar and not a set, and expressions may be undefined. How undefined expressions are treated is not fully explained [WK99, p.56]. And, oddly, the expression ***self.f.notEmpty*** is used to state that ***self*** is mapped by ***f***, even though ***self.f*** has at best a scalar value, and ***notEmpty*** is an operator only on collections [WK99, p.80].

Expressiveness

UML is generally more expressive than Alloy. Its datatypes include sequences, bags, strings and numbers. In its relational subset, however, it is less expressive than Alloy. Because there is no transitive closure operator, the Alloy constraint

```
no p | p in p.+parents
```

cannot be expressed in UML. As a workaround, the UML definition, which uses OCL for its well-formedness rules, attempts to axiomatize closure. For example, in [Rat97, Section 9.3], the following equation is given:

```
// The operation allPredecessors results in the set of all Messages that precede the current one.
allPredecessors : Set(Message);
allPredecessors = self.predecessor->union (self.predecessor.allPredecessors)
```

This does not have the desired effect, however; the operation may return the set of all messages and still satisfy its specification.

5 Discussion

Evaluation

In this paper I've attempted to show, by way of example, that Alloy can be as natural as UML, but at the same time rests on a firm semantic foundation. Developing a simple semantics and type system for Alloy has helped me simplify the language and root out ambiguities. Both have been 'implemented' in Fox, Alloy's analyzer: the semantics as a translation to boolean formulas, and the type system as a type checker.

Alloy's graphical notation has been taught twice to undergraduates in MIT's software engineering course 6170. Its textual notation has been used in several small case studies, including the design of a graph editor, the reverse engineering of a component of CTAS (a NASA air-traffic control system), and most recently in a demonstration that Fox could automate an analysis of COM described originally in Z [SSM97].

Alloy has many deficiencies, some noted in the discussion above. The most serious are the omission of relational formulas, which make operations hard to write (and might be remedied either by adding such formulas or by inferring frame conditions), and of sequences. Alloy also needs better structuring mechanisms. It lacks the generic schemas of Z, so the modeller cannot define polymorphic operators or constraints (eg, that a relation forms a tree). Also, it has no features for composing models (see for example, the notion of views in [Jac95] or frameworks in [DW98]).

Related Work

The Catalysis method of D'Souza and Wills [DW98] embodies a refinement of UML. Its graphical notation is close to Alloy's, and its textual notation offers infix variants of OCL's post-fix operators. The book offers perhaps the clearest and most rigorous explanation of UML's features. But Catalysis's contribution is primarily methodological, and the underlying semantics of UML is not addressed in detail.

Several schemes have been devised to translate object models to formal specifications, both as an exercise to expose semantic issues, and as a means of obtaining an analyable model. Larch is a popular target because its Shared Language provides a mathematical foundation that is not biased towards any style of specification structuring [BC95, BG98, HHK98]. Bickford and Guaspari's translation [BG98] addresses many of the complexities of UML, including subtyping and contexts. Their work also exposes a number of serious flaws in the definition of UML.

Z is another popular target, because its built-in datatypes—sets and relations–are well-suited to object modelling. Larch's algebraic operators, in contrast, are not a natural fit, and translation is rather indirect. UML classes are usually represented in Z with schema types [FBL97, FBLS97], following the object-oriented style of [Hall90]. More recently, work has begun on a formalization of UML in which semantic functions are defined in Z [AC98].

My work has a different aim from all of these. I have not attempted to account for the complexities of existing object modelling notations, but rather to devise a new language without the complexities. In particular, Alloy dispenses with subtyping and undefined expressions, and has a simple semantics and type system that can be described in a couple of pages (see Appendix A). I have followed this path for two reasons. First, I am unconvinced that the complexities of UML add significantly to its utility. And second, I am not optimistic about the possibilities of automated tool support for a language that was designed without analysis in mind.

Acknowledgments

Thanks to John Chapin, Michael Jackson, Barbara Liskov and Allison Waingold for their comments on earlier versions of Alloy, and to Martin Abadi for pointing out that Alloy's confusion of singleton sets and scalars has a fine pedigree [Kan97]. This research was funded in part by the National Science Foundation (under grant CCR-9523972), and by by the MIT Center for Innovation in Product Development (under NSF Cooperative Agreement Number EEC-9529140). Thanks also to the Visio Corporation for donating diagram drawing software to my class, and for helping in the development of an Alloy stencil.

References

[AC98] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop*, Ilkley, Electronic Workshops in Computing. Springer-Verlag, 1998.

[BC95] Robert H. Bourdeau and Betty H.C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, October 1995.

[CD94] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

[DW98] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With Uml : The Catalysis Approach*. Addison-Wesley, 1998.

[FBL97] Robert B. France, Jean-Michel Bruel and Maria M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. *Journal of Object Oriented Programming* (JOOP), 10(7), November/December 1997.

[FBLS97] 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), 1997.

[Hall90] Anthony Hall. Using Z as a Specification Calculus for Object-Oriented Systems. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, eds., *VDM and Z: Formal Methods in Software Development*, Lecture Notes in Computer Science, Volume 428, pp. 290–381, Springer-Verlag, New York, 1990.

[Hay93] Ian Hayes. *Specification Case Studies*. Prentice Hall, 1993.

[HHK98] Ali Hamie, John Howse and Stuart Kent. Interpreting the Object Constraint Language. *Proceedings of Asia Pacific Conference in Software Engineering*, IEEE Press, 1998.

[Jac95] Daniel Jackson. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 4, October 1995, pp. 365–389.

[Jac98] Daniel Jackson. An Intermediate Design Language and its Analysis. *Proceedings of ACM SIGSOFT Foundations of Software Engineering*, Orlando, Florida, 1998.

[JD96a] Daniel Jackson and Craig A. Damon. *Nitpick Reference Manual*. CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JD96b] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[Kan97] Akihiro Kanamori. The mathematical import of Zermelo's well-ordering theorem. Bulletin of Symbolic Logic, Volume 3, Issue 3, September 1997, pages 281 - 311.

[MJa95] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[Par95] David Parnas. A Logic for Describing, not Verifying, Software. Erkenntnis (Kluwer), Volume 43, No. 3, November 1995, pp. 321–338.

[Rat97] Rational Inc. The Unified Modeling Language. see http://www.rational.com.

[Rum91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Saa97] Mark Saaltink. Domain Checking Z Specifications. 4th NASA LaRC Formal Methods Workshop, September 1997.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second ed, Prentice Hall, 1992.

[SSM97] K.J. Sullivan, J. Socha and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. *Proceedings of the International Conference on Software Engineering* (ICSE97), Boston, Massachusetts, May 1997.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

frame ::= *decl* * *formula*

decl ::= *set-comp* : *domain* | *relation-comp* : *domain* -> *domain*
| *indexed-comp* [*domain*] : *domain* -> *domain*

formula ::= *not formula* | *formula* && *formula* | *all var* : *domain BAR formula* | *expr in expr*

expr ::= *set-comp* | *expr* + *expr* | *expr* – *expr* | *expr* & *expr* | *expr* . *suffix*

suffix::= *qualifier* | *qualifier* . *suffix*

qualifier ::= *relation* | + *qualifier* | ~ *qualifier*

relation ::= *relation-comp* | *indexed-comp* [*var*]

set-comp ::= *identifier* | *identifier PRIME*

relation-comp ::= *identifier* | *identifier PRIME*

indexed-comp ::= *identifier* | *identifier PRIME*

variable ::= *identifier*

domain ::= *identifier*

Figure A1: Syntax of Alloy Kernel

Appendix A: Kernel Syntax and Semantics

Alloy is based on a much smaller kernel language that is technically equal in expressive power to the full language (but much less convenient to use). We shall give the semantics in terms of the kernel, and then explain how the additional features of the full language can be regarded as syntactic sugar.

The top-level syntactic entity in the kernel language is a *frame*. Frames are just formulas with accompanying declarations, and form the basis of the schemas of a full specification.

Syntax

An abstract syntax for the kernel is given in Figure A1. BAR and PRIME are non-terminals representing the vertical bar and prime mark respectively.

A frame consists of a series of declarations of state components, followed by a formula. Formulas are built from the standard logical operators, and universal quantifiers, and from just one kind of elementary formula. An expression is either the name of set component, or a composite expression using standard set operators, or a navigation expression formed from an expression and a suffix. A suffix is a sequence of qualifiers; a qualifier is the name of a relation, with transitive closure and transpose applied optionally as prefix operators. A relation is named either with a relation component, or with an indexed relation component, and its index, which must be a variable. Finally, the state components may optionally be marked with a prime.

Typing

Alloy is strongly typed. A formula is typed within an environment that maps identifiers—namely variables and state components—to types. Types take three forms. Expressions (and set components) are given set types; the judgment

$$\frac{E[\text{co (d0)} : ty\ \text{(d0)}, \text{co (d0)'} : ty\ \text{(d0)}] \vdash d1 \ldots dn\ f}{E \vdash d0\ d1 \ldots dn\ f}$$

$$\frac{E \vdash f, E \vdash g}{E \vdash f\ \&\&\ g}$$

$$\frac{E \vdash f}{E \vdash not\ f}$$

$$\frac{E[v : T] \vdash\ f}{E \vdash all\ v : T \mid f}$$

$$\frac{E \vdash e1 : T, E \vdash e2 : T}{E \vdash e1\ in\ e2 : T}$$

$$\frac{E \vdash e1 : T, E \vdash e2 : T}{E \vdash e1 + e2 : T}$$

$$\frac{E \vdash e1 : T, E \vdash e2 : T}{E \vdash e1 \& e2 : T}$$

$$\frac{E \vdash e1 : T, E \vdash e2 : T}{E \vdash e1 - e2 : T}$$

$$\frac{E \vdash e : S, E \vdash s : S \rightarrow T}{E \vdash e.s : T}$$

$$\frac{E \vdash q : S \rightarrow T}{E \vdash \sim q : T \rightarrow S}$$

$$\frac{E \vdash q : S \rightarrow T}{E \vdash +q : T \rightarrow S}$$

$$\frac{E \vdash r : S \rightarrow T \rightarrow V, E \vdash v : S}{E \vdash r[v] : T\ ?V}$$

$$\frac{E \vdash q : S \rightarrow T, E \vdash s : T \rightarrow V}{E \vdash q.s : S \rightarrow V}$$

Figure A2: Type rules for Alloy Kernel

$$E \vdash e : T$$

says that, in environment E, expression e denotes a set of elements of primitive type T. Suffixes (and relation components) are given relation types; the judgment

$$E \vdash s : S \rightarrow T$$

says that the suffix s denotes a relation from primitive type S to primitive type T. These judgments follow directly from the declarations. Lastly, an indexed relation component r declared as

$$r[S] : T \to V$$

results in the judgment

$$E \vdash r : S \to T \to V$$

saying that r denotes a function that maps objects from the primitive type S to relations from T to V. In the type $S \to T$, we shall refer to S as the *left-type* and T as the *right-type*. In the indexed type $S \to T \to V$, S is the *index-type*, T the *left-type* and V the *right-type*. Note that there are no scalar types; a scalar is just a singleton set. Also, types may not be nested: S and T may not themselves be types.

Formulas do not have types, but must be type checked. To represent the successful typing of the formula f in the context of the declarations $d0$, $d1$, etc, we use the judgment

$$E \vdash d0\, d1 \ldots f$$

Declarations and quantifiers extend the environment; we write $E[x1 : t1, x2 : t2, \ldots]$ to denote the environment that is like E but which additionally assigns the type $t1$ to the variable or component $x1$, $t2$ to $x2$, etc. For a declaration d, $co(d)$ and $ty(d)$ are the component and type respectively. The environment $E[\mathsf{co}\ (\mathsf{d}) : ty\ (\mathsf{d}), \mathsf{co}\ (\mathsf{d})' : ty\ (\mathsf{d})]$ is E extended by an assignment of the type of d to its component and to its component primed.

The type rules are shown in Figure A2 in standard form as an inference system. Each rule has a hypothesis (above the line) and a consequent (below); when the hypothesis holds, the consequent may be inferred. For example, the rule

$$\frac{E \vdash e1 : T,\ E \vdash e2 : T}{E \vdash e1 + e2 : T}$$

says that if expressions $e1$ and $e2$ can be shown both to have type T, then the compound expression $e1 + e2$ has type T also. Note that, throughout, S and T denote primitive types and not arbitrary type expressions. So a hypothesis of the form $e : T$ does not mean that e has some arbitrary type T, but that e denotes a set of elements drawn from the primitive type T.

Semantics

The semantics of an expression or formula is defined in the context of an assignment that maps variables, components and domains to values. The value assigned to x by an assignment A is written $A[x]$, and is called the *denotation* of x. Since top-level formulas may have no free variables, they are interpreted with assignments that map only components and primitive types.

Values are constructed from a universe $\mathcal{E}$ of objects. For a given collection of formulas and declarations, an assignment A is well formed if it maps

1 each primitive type appearing in a declaration to a set of objects:

$$A[T] : \mathbb{P}\, \mathcal{E}$$

2 each declared set component to a set of objects;

$$A[s] : \mathbb{P}\, \mathcal{E}$$

3 each declared relation component to a set of pairs of objects;

$$A[p] : \mathbb{P}\, \mathcal{E} \times \mathcal{E}$$

4 each declared indexed relation component to a function from objects to sets of pairs of objects.

$$A[p] : \mathcal{E} \to \mathbb{P}\, \mathcal{E} \times \mathcal{E}$$

(Here $X \to Y$ denotes the partial function from X to Y.)

An assignment must also be well typed. That is, the value associated with a component must be consistent with the value associated with the constituents of its type:

1 the denotations of distinct primitive types are disjoint;

$$A[S] \cap A[T] \neq \emptyset \Rightarrow S = T$$

2 the denotation of a set component is a subset of the denotation of its primitive type;

$$E \vdash s : T \Rightarrow A[s] \subseteq A[T]$$

3 each pair in the denotation of a relation draws its first (second) element from the denotation of the left-type (right-type) of the relation's type;

$$E \vdash p : S \rightarrow T \Rightarrow A[p] \subseteq A[S] \times A[T]$$

4 the denotation of an indexed relation component is a function mapping objects belonging to the denotation of its index-type to a set of pairs whose first (second) elements are drawn from the denotations of the left (right) types:

$$E \vdash p : S \rightarrow T \rightarrow V \Rightarrow A[p] \in A[S] \rightarrow (\mathbb{P}\, A[T] \times A[V])$$

5 the denotation of a variable is a singleton set:

$$E \vdash v : T \Rightarrow \exists a.\, A[v] = \{a\} \wedge a \in A[T]$$

In the semantics, different assignments are used to interpret different parts – ie, subformulas, expressions, etc – of the formula. For a top-level formula, an assignment will not map any variables; within a quantified formula, an assignment will map all free variables. For any part of formula, the set of assignments that is well typed is determined using the type environment in which the type of the part itself was judged.

Figure A3 gives the semantics, in which the meaning of a formula is defined compositionally in terms of its parts. A separate semantic function is defined for each syntactic category.

Now, finally: the meaning of a kernel frame, consisting of a series of declarations and a formula *f*, is the set containing all assignments *A* that are well typed (as described above), and for which *F*[*f*]*A* is true. As we shall see in the next section, a frame might correspond to a collection of state invariants, in which case its meaning is the set of states satisfying the invariants. Alternatively, it may correspond to an operation, in which case its meaning is a set of state transitions. And finally, it may represent the negation of an assertion claimed to be true, in which case its meaning is a set of counterexamples: states if the assertion is about states, and transitions if about transitions.

Expressiveness

Since the kernel language represents the essence of the full language, it can act as a surrogate for theoretical purposes. I have yet to conduct a serious investigation into Alloy's expressive power, but a few key properties are clear.

Alloy is a first-order language: there are no relations over relations, for example, or quantifiers over sets or relations. On the other hand, the language is more expressive than first-order logic with at most dyadic predicates, since it includes transitive closure, which cannot be axiomatized in a first-order fashion. Since tuples can be modelled with extra primitive types, and appropriate projection functions, an argument could be made (similar to that made by Tarski for the relational calculus) that Alloy subsumes first-order logic. At any rate, Alloy is at least as powerful as its predecessor NP (see [Jac98]), and is less powerful than Z. It should be straightforward to show that Alloy is at least as expressive as Tarski's relational calculus (since any formula about relations can be expressed in Alloy as a formula about how the relations map their elements), and is thus undecidable.

Semantic functions

$F : Formula \rightarrow Assignment \rightarrow Bool$

$E : Expr \rightarrow Assignment \rightarrow \mathsf{P}\ \mathsf{E}$

$S : Suffix \rightarrow Assignment \rightarrow \mathsf{P}\ \mathsf{E} \times \mathsf{E}$

Formulas

$F\ [f\ \&\&\ g]\ A = F\ [f]\ A \wedge F\ [g]\ A$

$F\ [\text{not } f]\ A = \neg F\ [f]\ A$

$F\ [\text{all v: T} \mid \text{f}]\ A = \wedge \{F\ [f]\ (A \oplus v \mapsto a) \mid a \in A\ [\ T\ ]\ \}$

Expressions

$E\ [v]\ A = A\ [v]$

$E\ [s]\ A = A\ [s]$

$E\ [d + e]\ A = E\ [d]\ A \cup E\ [e]\ A$

$E\ [d\ \&\ e]\ A = E\ [d]\ A \cap E\ [e]\ A$

$E\ [d - e]\ A = E\ [d]\ A \setminus E\ [e]\ A$

Suffixes

$S\ [p[v]]\ A = E[p]\ (E[v]\ A)\ A$

$S\ [\sim p]\ A = \{\ (y, x)\ |(x, y) \in E\ [p]\ A\}$

$S\ [+p]\ A = \{\ (x, z)\ |\exists y1, y2, \ldots, yn.\ (x,y1) \in E\ [p]\ A \wedge (y1,y2) \in E\ [p]\ A \wedge \ldots \wedge (yn,z) \in E\ [p]\ A\}$

$S\ [p.s]\ A = \{(x, z) \mid \exists y.\ (x, y) \in E\ [p]\ A \wedge (y, z) \in E\ [s]\ A$

FigureA3: Semantics of Kernel

Appendix B: Full Language

Alloy is a small language, but is still much larger than the kernel In this section, I'll illustrate each respect in which the full language extends the kernel. All the extensions are defined as straightforward syntactic sugarings, so no new semantic notions are needed.

Structuring Mechanisms

Formulas are organized into schemas that, in the context of checking, play different roles. For our purposes here, however, each schema, whatever its role, defines a set of assignments.

Operator Shorthands

The first category of extensions is trivial. It adds the missing logical operators (disjunction and implication), existential quantification, equality of expressions, and reflexive-transitive closure for qualifiers:

shorthand	*expanded form*
f || g	*not (not f && not g)*
f -> g	*not (f && not g)*
some v: d | f	*not all v: d | not f*
d = e	*d in e && e in d*
*e.*q.s*	*(e.+q) + e).s*

Quantifier Shorthands

Some convenient quantifiers are added, shown with their expansions in the table below. They assert there is no value satisfying the formula (no), at most one (sole), and exactly one (one).

shorthand	*expanded form*
no v: d | f	*all v: d | not f*
sole v: d | f	*all v1: d | all v2: d | (some v: d | v = v1 && f) &&(some v: d | v = v2 && f) -> v1 = v2*
one v: d | f	*(sole v: d | f) && (some v: d | f)*

The traditional shorthand for nested quantifiers is available:

all a, b: d | f

is short for

all a: d | all b: d | f

Implicit Types

Types of components and variables are implicit in Alloy, and are automatically inferred. An Alloy model begins by declaring a collection of set components marked by the keyword *domain*. With each of these set components, called domains, a primitive type of the same name is implicitly associated. Subsequent component declarations may then refer to these domains, and to each other (so long as there are no forward references). For example, the declarations

```
domain { Person }
state {
 Man, Woman : Person
  wife : Man -> Woman
}
```

introduce four state components and a primitive type *ty(Person)*; the equivalent kernel declarations would be

Person : *ty(Person)*
Man : *ty(Person)*
Woman : *ty(Person)*
wife : *ty(Person)* -> *ty(Person)*

Note that the *ty(Person)* appearing on the right here is a type name, and is used in type checking; the *Person* appearing on the left is a set component. The declarations in the full language express additional properties: that the sets *Man* and *Woman* are subsets of the set *Person*, and that the *wife* relation maps elements of *Man* to elements of *Woman*.

This scheme brings several advantages. The primary motivation is to match the intuitive semantics of graphical object models, in which boxes without parents in the diagram represent sets that are assumed to be disjoint, but which, unlike the sets denoted by the given types of Z, are not constant. It dispenses with the need to associate given types explicitly with these sets; allows declarations to include simple constraints; and makes type checking only marginally harder and no less effective.

Types are also implicit in quantified formulas. The formula *(Q v | f)*, where *Q* is a quantifier, is short for *(Q v: t | f)*, where *t* is the primitive type of *v* inferred from *f*, according to the rules of Figure A2. In certain pathological formulas, such as *(all v | v = v)*, the type of the variable is under-constrained, and the formula is deemed to be ill-typed.

Quantifier Bounds

Quantified variables may be bounded; *all v: e | f*, for example, is taken to be short for

```
all v | v in e -> f
```

There is also implicit bounding. According to the semantics, each variable can only take on a value from the denotation of its primitive type. It turns out, however, to be useful to make the implicit bound stronger, and to regard *all v | f* as short for

```
all v | v in dv -> f
```

where *dv* is the domain associated with the primitive type of *v*.

There are two rather technical reasons for doing this. First, the domains represent the world of atoms known to the object model. When we write a formula such as

```
all p | p in (Woman + Man)
```

we expect it to mean that every *Person* is a *Man* or a *Woman*. Were the bound to be the primitive type *Person* and not the domain *Person*, this formula would not hold for the state in which

```
ty (Person) = {alice, bob, carol}

Person = {alice, bob}

Man = {bob}

Woman = {alice}
```

simply because the primitive type *ty*(*Person)* includes an extra element *carol*.

Second, we would like formulas to be 'scope-monotonic'. An assignment is said to be within a scope of *k* if the denotation of each primitive type is a set containing no more than *k* elements. A formula is scope-monotonic if, whenever it has a satisfying assignment in a scope *k*, it also has a satisfying assignment in scope *k+1*. This property turns out to be crucial in implementing a checking tool that evaluates a formula within a finite scope. Without it, the bizarre situation can arise in which increasing the scope and thus considering more assignments results in fewer solutions.

Quantifiers let Alloy dispense with set constants. The formula *Q e*, where *Q* is a quantifier and *e* is an expression, is short for

```
Q v | v in e
```

So *some Woman* asserts that the set *Woman* is non-empty; *no Man & Woman* asserts that the intersection of *Man* and *Woman* is empty; and *all Man* asserts that every *Person* is a *Man*.

Multiplicity & Domain Constraints

A multiplicity marking in the declaration of a set component bounds the size of the set by adding conditions as follows:

declaration	*condition*	*informal interpretation*
s : t !	*one s*	*s is scalar*
s : t ?	*sole s*	*s is optional*
s : t +	*some s*	*s is non-empty*

Multiplicity markings in the declaration of a relation component give rise to conditions on its value as a whole (rather than conditions on the pairs individually). The left- and right-types are treated independently; conditions are added for the multiplicity markings of each in turn, according to this table:

declaration	*condition*	*informal interpretation*
r : s -> t ?	*all a: s | sole a.r*	*r is a partial function*
r : s -> t !	*all a: s | one a.r*	*r is a total function*
r : s -> t +	*all a: s |some a.r*	*r is total*
r : s ? -> t	*all a: t | sole a.~r*	*r is injective*
r : s ! -> t	*all a: t | one a.~r*	*r is injective and surjective*
r : s+ -> t	*all a: t |some a.~r*	*r is surjective*

The conditions for an indexed relation are the same, but wrapped appropriately. So the condition obtained from

r [s] :t -> ? v

for example, is

all x: s | all a:t | sole a.r[x]

A collection of sets may be declared to be disjoint or to form a partition. These give rise to the obvious constraints.

Mutability

Mutability constraints are basic temporal constraints, so they only come into play when state transitions are considered. A proper treatment of operations is beyond the scope of this paper, but for now a simple example should suffice. The schema

op add (e: T!) {s′ = s + e}

declares an operation whose effect is to insert *e*, the argument of the operation, into the set *s*.

Mutability constraints add frame-conditions to all operations. A set component may be marked as being fixed or static; components with associated primitive types (ie, those declared in the domains schema) are by definition static. A fixed set is constant; its value never changes, and if an element is in the set before a transition, it is in the set after. For a set *s* of primitive type *t*, this condition may be written as

all e : t | e & s = e & s′

A static set, on the other hand, maintains its containment of an element only when that element is not created or destroyed in the transition; this condition may be written

all e : d & d′ | e & s = e & s′

where *d* is the domain associated with the primitive type *t* of *s*.

When the right-type of a relation is labelled as static, thus

r : A -> static B

the relation is said to be right-static, and the set of elements in the right-type mapped to by with an element of the left-type is fixed so long as that element is not created or destroyed:

all a : A & A′ | a.r = a.r′

When the left-type is marked as static

r : static A -> B

the relation is left-static, and the set of elements that map to a given element in the right-type is fixed:

all b : B & B' | b.~r = b.~r'

The combination of multiplicity and mutability constraints allow enumerations to be declared conveniently without extending the language. For example,

partition Red, Blue, Green : fixed Colour !

says that *Red*, *Blue* and *Green* represent scalars (from the multiplicity), that they partition the set *Colour*, and that their values do not change.

Partial Functions

In all specification languages in which functions can be represented, a fundamental question arises: how to give meaning to the formula $f(x) = y$ in which the function f is applied to a value x that is outside its domain. This question has been the subject of debate for almost a century; it is connected with the old problem, in logic, of whether "The King of France is dead" is true if France has no king.

Specification languages have taken several approaches, each with its merits. One approach, taken by Larch and advocated by Schneider [] and others, is to model all functions as if they were total. This approach is simple and clear, and preserves Leibniz's law—that from $x = y$ one can infer $f(x) = f(y)$. But it is unsuitable for an object modelling language, since it makes an unnatural distinction between functions and relations, functions being required to carry with them the set representing the domain.

Another common approach is to introduce a notion of undefinedness. This has many variants, but it always introduces complications: a three-valued logic (in VDM), a semantics of multiple interpretations (in Z), ad hoc distinctions between operators (in UML), and inevitably the loss of Leibniz's law. In practice, it seems to confuse specifiers; Saaltink has analyzed a large body of published Z specifications and found almost all (including one of mine!) to be faulty in their use of partial functions [Saa97]. Even the innocuous-looking assertion $\#s = 3$, that the set s has three elements, is undefined in Z unless the type declaration of s says explicitly that s is finite.

A simpler approach is advocated by Parnas [Par95] and was adopted in Alloy's predecessor, NP. An elementary formula in which a function is applied outside its domain is deemed to evaluate to false. In this approach, we can infer from $f(x) = y$ that x is in the domain of f, so the formula

$\exists x, y. f(x) = y$

which is undefined in Z, has the expected meaning, namely that some element is mapped by f. Unfortunately, however, $f(x)=f(x)$ is sometimes false. Moreover, the specifier must know which formulas are regarded as elementary: if x is outside the domain of f, $f(x) \neq y$ will be true if $\neq$ is an elementary operator and false otherwise.

Alloy sidesteps the partial function problem by eliminating scalars, and treating functions no differently from other relations. Leibniz's law holds; there is no special treatment of elementary formulas; and no need for undefinedness. Most importantly, the guards required in languages such as Z can be dropped (examples of this are given below). The one price paid, it seems, is that membership tests can have an unexpected meaning. The Alloy formula *(a.f in s)* will be true if a is outside the domain of f, since *a.f* then evaluates to the empty set, which is a subset of any set. For this reason, Alloy provides a special operator for membership:

a.f !in s

is short for

a.f in s && some a.f

which is false whenever a is not in f's domain.

Indexed Relations

Indexed relations are part of the kernel, but their rationale is better discussed in the context of the full language. A declaration

```
r [X] : A -> B
```

declares *r* to be an indexed collection of relations. For each value *x* of the domain *X*, there is a relation *r[x]* from *A* to *B*. For example, we might declare

```
met [Loc] : Person -> Person
```

where *met [loc]* relates *p* to *q* when *p* met *q* in the location *loc*. To avoid the partial function problem, the semantics associates a relation with every value of the index in the domain; also, the index is required to be a variable, so that the question of what *met[e]* means when *e* evaluates to a non-singleton-set does not arise. Here is an example of a formula:

```
all loc, p, q | p in met[loc].q -> q in met[loc].p
```

which says that the notion of meeting is symmetrical.

This scheme, which is based on the notion of qualifiers in OMT, is nice because it does not perturb the fundamentally binary nature of the notation. It is not always convenient, however, and I am working on an extension that would allow arbitrary tuples to be formed.