# Alloy: A Lightweight Object Modelling Notation

Daniel Jackson
Laboratory for Computer Science
Massachusetts Institute of Technology
November 27, 2001

## Abstract

Alloy is a little language for describing structural properties. It offers a declaration syntax compatible with graphical object models, and a set-based formula syntax powerful enough to express complex constraints and yet amenable to a fully automatic semantic analysis. Its meaning is given by translation to an even smaller (formally defined) kernel. This paper presents the language in its entirety, and explains its motivation, contributions and deficiencies.

## 1 Introduction

What is the smallest modelling notation that can express a useful range of structural properties, is easy to read and write, and can be analyzed automatically? This paper describes an attempt to answer this question. Alloy is an experimental language with a small syntax, built on an even smaller kernel. The kernel has a precise semantics, and is expressive enough to capture complex properties, while remaining amenable to efficient analysis.

Almost all recent development methods factor out the structural aspect of a software system for separate description, usually called the 'object model'. Alloy supports the description of basic structure (graphically, or as textual declarations), as well as more intricate constraints and operations describing how structures change dynamically (both expressed as logical formulas). It thus incorporates not only the object model, but also the 'operation model' of Fusion [8], or the 'behaviour model' of Catalysis [13], and is comparable to the Object Constraint Language [63] of UML [53]. Alloy is not for describing dynamic interactions between objects, nor for describing syntactic structure in an implementation, such as the class hierarchy and packaging.

Alloy is amenable to a fully automatic semantic analysis that can provide checking of consequences and consistency, and simulated execution [31,40]. To gain 'executability', Alloy does not sacrifice abstraction: it can generate sample transitions of an operation described implicitly, using negation and conjunction [33].

Alloy and its predecessor NP [30,34] have been used to model and analyze a variety of artifacts, including architectural frameworks [12,37], a mobile internet protocol [39], a naming scheme [43], the UML core metamodel [61], and a message filtering device [65].

Alloy's starting point is Z [58], an elegant and powerful language with a particularly simple mathematical foundation. It selects from Z those features that are essential for object modelling, and incorporates a few constructs that are ubiquitous in more recent (but less formal) notations. The semantics of Alloy thus bridges the gap between Z and object models, and shows how to give simple and robust meaning to widely used forms, such as navigation expressions and object model diagrams.

Semantics has been a vital design tool in this project, ensuring that the language elements are clear and well defined, and can be composed flexibly without unexpected interactions. In most other attempts to combine the benefits of formal and informal notations, semantics has been used instead to explain and make sense of existing elements. Although this explanatory approach can remedy the imprecision and ambiguity of an informal notation, it does not make it simpler, and may even sanction its complexity. Alloy recognizes value in current informal notations not in their popularity per se, but in the particular features that appear repeatedly because of their elegance and utility.

The paper starts with an example: an Alloy model of file system structure (Section 2). The features of Alloy are explained informally, and the kinds of analysis that can be performed are illustrated. It presents the kernel, with its semantics, and explains informally how the full language is translated to it (3). The graphical sublanguage is explained as a variant of the textual language (4). The rationale for Alloy is then given in some detail: why Alloy is based on Z, in preference to UML in particular, but why Z alone was not deemed adequate (5). Experiences with Alloy, and its known deficiences, are summarized (6). The paper closes with related work (7) and a brief summary (8). A full grammar of Alloy appears in the appendix.

## 2 Example

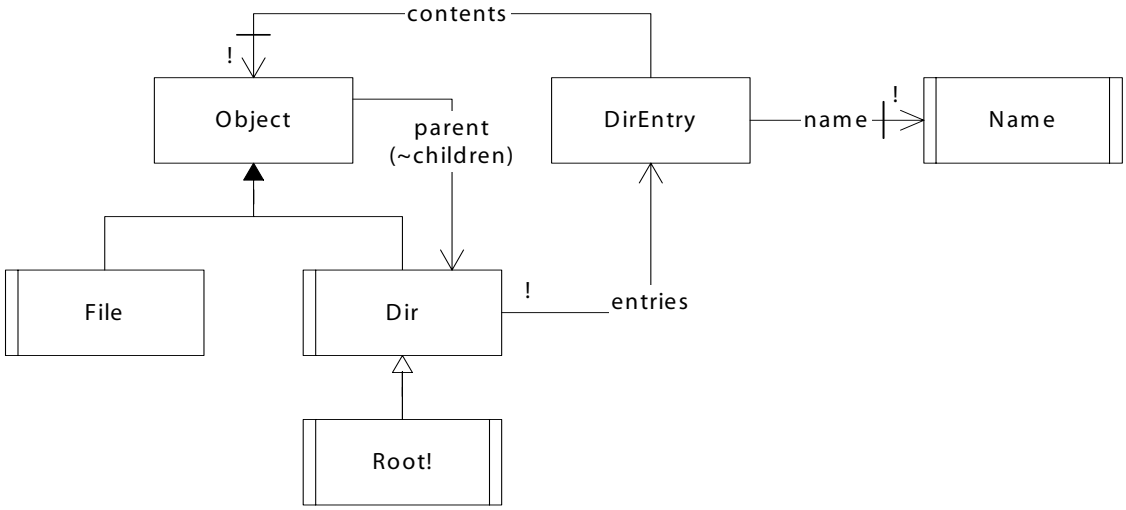
The Alloy model of Figure 1 is a partial description of a file system (adapted from a model given in [46]). Our discussion will focus on the textual version; the syntax for the diagram, which corresponds to the *domain* and *state* paragraphs of the text, is explained later (in Section 4).

### 2.1 Specification

The model is divided into paragraphs. The *domain* paragraph declares sets that represent the coarsest classification of atoms, in this case into file system objects, directory entries and names. Because objects and directory entries can be created and deleted, these sets may change, and are viewed as components of the state. We will not be concerned with the creation and deletion of names, though, preferring to think of them being drawn from a fixed pool. So the set of names is declared to be *fixed*. Each domain implicitly introduces a primitive type. It will therefore be a type error, for example, to form the union of two distinct domains, or of their subsets.

The *state* paragraph declares the remaining state components. *File* and *Dir* are sets that at any time partition the set *Object* of file system objects. These sets are *static*, which means that they represent fixed classifications of objects. Although files and directories may be created and destroyed, a file may not become a directory or vice versa. *Root* is a particular fixed directory, the exclamation mark indicating that it represents a set of exactly one element. The relation *entries* maps directories to their directory entries; the exclamation mark again means exactly one, in this case that each entry is in one directory. The relations *name* and *contents* map directory entries to their names and contents, and are static, meaning that during the lifetime of an entry its name and contents may not change. Lastly, *parent* is a relation that maps an object to the directories it belongs to, and *children* is defined to be its transpose.

The *definition* paragraph defines *parent* in terms of other state components. The formula is read:




```
model FileSystem {
    domain {Object, DirEntry, fixed Name}
    state {
        partition File, Dir : static Object
        Root: fixed Dir!
        entries: Dir! -> DirEntry
        name: DirEntry -> static Name!
        contents: DirEntry -> static Object!
        parent (~children) : Object -> Dir
        }
    def parent { all o | o.parent = o.~contents.~entries }
    inv UniqueNames {all d | all e1, e2: d.entries | e1.name = e2.name -> e1 = e2}
    inv Parents {
        no Root.parent
        all d: Dir - Root | one d.parent }
    inv Acyclic {no d | d in d.+parent}
    inv Reachable {Object in Root.*children}
    cond TwoDeep {some Root.children.children}
    assert FileHasEntry {all f: File | some d | f in d.entries.contents}
    assert AtMostOneParent {all o | sole o.parent}
    op NewDirEntries (d: Dir, es: DirEntry') {
        no es & DirEntry
        d.entries' = d.entries + es
        all x: Dir - d | x.entries' = x.entries }
    op Create (d: Dir!, o: Object'!, n: Name) {
        n !in d.entries.name
        some e: DirEntry' | NewDirEntries (d, e) && e.contents' = o && e.name' = n }
    assert EntriesCreated {all d: Dir, e: DirEntry' | NewDirEntries (d,e) -> DirEntry' = DirEntry + e}
    assert CreateWorks {all d, o, n | Create (d,o,n) -> o in d.children'}
    }
```

*Figure 1: Sample Specification*

for all objects *o*, to find the parents of *o*, follow the *contents* relation backwards (thus obtaining the set of directory entries for which this object is the contents), and then follow the *entries* relation backwards too (obtaining the set of directories that contain these entries). Note that the quantification has an implicit bound: the type of the variable *o* is inferred to be that of the domain *Object*, and *o* is constrained to be drawn from that domain.

A series of *invariants* follow. *UniqueNames* says that any two distinct entries *e1* and *e2* of the same directory *d* must have different names. *Parents* says, in two formulas implicitly conjoined, that the root has no parent but all other directories have one parent. *Acyclic* says that no directory is an ancestor of itself. *Reachable* says that every object is reachable from the root: *Root.*children* is the set of objects obtained by following the *children* relation zero or more times from *Root*, and is required to contain the set of all objects.

There is no special notion of scalars in Alloy. A scalar is declared as a set with one element; the operator *in*, whose semantics is that of subset, may sometimes be read 'is an element of' (as in *Acyclic*), and sometimes 'is a subset of' (as in *Reachable*). Quantifier keywords are applied to expressions to denote their cardinality: *no X, some X, one X* and *sole X* mean respectively that the set *X* has no elements, some elements, one element and at most one element.

The *condition* that appears next, *TwoDeep*, says that there are some children of the children of the root. Unlike an invariant, it is not expected always to hold; it was included for simulation, to check that the invariants did not rule out structures more than one level deep.

Two *assertions* follow. *FileHasEntry* says that each file is the contents of an entry in some directory. *AtMostOneParent* says that each object has at most one parent. These assertions are intended to be redundant; analysis will check that they follow from the invariants.

Only two *operations* are included here. *NewDirEntries* adds a set of entries *es* to a directory *d*. The appearance of *DirEntry* in the argument list is primed because the entries *es* are created and will therefore belong to the domain *DirEntry* after but not before execution. The three constraints in the body of the operation say that: (1) the sets *es* and *DirEntry* have no intersection; (2) the entries of *d* are extended by *es*; and (3) the set of entries associated with other directories is unchanged. The *Create* operation takes a directory, an object and a name. It creates a fresh entry in that directory with the given name and with the object as contents. The name is required to be distinct from all names of entries already in the directory. This operation is defined in terms of the other.

The last two assertions record intended properties of the operations. *EntriesCreated* says that executing *NewDirEntries* for a given entry adds just that entry to the set of entries (and removes none). *CreateWorks* says that executing *Create* makes the given object a child of the given directory.

## 2.2 Analysis

Alloy supports two kinds of analysis: *simulation*, in which the consistency of an invariant or operation is demonstrated by generating a state or transition, and *checking*, in which a consequence of the specification is tested by attempting to generate a counterexample.

The power of declarative specification, in which negation and conjunction are used freely, comes with two risks: over- and under-constraint. Simulation helps catch errors of overconstraint, by reporting, contrary to the user's intent, that no instance exists within the finite bounds of a given

'scope', or by showing instances with an unintended structural property. Checking helps catch errors of underconstraint, by showing instances that are acceptable to the specification but which violate an intended property.

Together, the two analyses enable an incremental process of specification. One starts with a minimal model, and performs a variety of simulations to detect overconstraint. Intended consequences are formulated, with counterexamples suggesting additional constraints to be added to the specification. This process helps produce a specification that has the desired properties and no more.

Alloy's analyses have been used in other ways too. One can formulate as checks the equivalence of two formulations of a property, thus evaluating a purported simplification. This strategy was used to simplify a model of the query interface mechanism of Microsoft COM [37]. One can also use the analysis in the style of traditional model checking, in an attempt to find subtle errors in existing designs. This strategy was used to expose flaws in the design of a resource discovery system [43].

Analysis of the file system model (Figure 1) might start with an attempt to simulate the state invariants (probably before any operations have been written). Alloy's tool [40] lets the user select a paragraph; to find a state satisfying all invariants, one selects the state paragraph. The tool displays the state in which the only object is the root directory. All sets and relations are empty, except for the domain *Object* and the sets *Dir* and *Root* which all contain the single atom *O1*. To make the results more readable, the tool will actually the use the name *Root* in place of *O1*:

```
Analyzing state ...
Scopes: DirEntry(3), Name(3), Object(3)
Conversion time: 0 seconds
Solver time: 0 seconds
Instance found:
Domains:
 DirEntry = {}
 Name = {}
 Object = {Root}
Sets:
 Dir = {Root}
 File = {}
Relations:
 children = {}
 contents = {}
 entries = {}
 name = {}
 parent = {}
```

Suppose we had erroneously declared contents as

```
contents: DirEntry+ -> static Object!
```

saying that each object is the contents of at least one entry. The simulation would now report that no instances are found, exposing the overconstraint. The default scope assigns 3 elements to each primitive type, so this does not imply that no instance exists, but rather that none can be constructed with at most most 3 objects, 3 entries and 3 names.

Not all overconstraint eliminates all instances, of course. Perhaps the specification rules out any file system except the empty one. To guard against partial overconstraint, one simulates a variety of conditions. *TwoDeep* illustrates that the specification allows a system that is two levels deep:

```
Analyzing TwoDeep ...
Scopes: DirEntry(3), Name(3), Object(3)
Conversion time: 0 seconds
Solver time: 0 seconds
Instance found:
Domains:
 DirEntry = {D0,D1}
 Name = {N0}
 Object = {Root,O0,O1}
Sets:
 Dir = {Root,O0,O1}
 File = {}
Relations:
 children = {Root -> {O1}, O1 -> {O0}}
 contents = {D0 -> O1, D1 -> O0}
 entries = {Root -> {D0}, O1 -> {D1}}
 name = {D0 -> N0, D1 -> N0}
 parent = {O0 -> {O1}, O1 -> {Root}}
```

Simulation sometimes exposes underconstraint also. With the *Acyclic* invariant omitted, *TwoDeep* gives an instance in which the root directory is the contents of a directory entry.

Checking the assertion *FileHasEntry* gives no counterexample. Because of the finite scope, we cannot conclude that the assertion holds. But by increasing the scope, we can gain greater confidence. Although only small scopes are feasible, it is usually possible to complete an analysis for a scope sufficient to find most problems that arise in practice. A scope of 6, for example, is exhausted in about 10 seconds on a modestly equipped PC, and it seems unlikely that there are properties of the file system that can only be illustrated in larger scopes.

Analysis tends to terminate much sooner when a counterexample is found. *AtMostOneParent* is invalid, and gives a counterexample in a scope of 3:

```
Analyzing AtMostOneParent ...
Scopes: DirEntry(3), Name(3), Object(3)
Conversion time: 0 seconds
Solver time: 0 seconds
Counterexample found:
Domains:
 DirEntry = {D0,D1,D2}
 Name = {N0,N2}
 Object = {Root,O0,O1}
Sets:
 Dir = {Root,O0,O1}
 File = {}
Relations:
 children = {Root -> {O0,O1}, O1 -> {O0}}
 contents = {D0 -> O1, D1 -> O0, D2 -> O0}
 entries = {Root -> {D0,D2}, O1 -> {D1}}
 name = {D0 -> N2, D1 -> N2, D2 -> N0}
 parent = {O0 -> {Root,O1}, O1 -> {Root}}
```

Operations are simulated in the same way as invariants. *Create*, as expected, shows an object being added as a child of the root in the empty file system. The assertion *CreateWorks* is valid, but *EntriesCreated* is invalid, sincethe set *Dir* may change. A directory in *Dir'* but not *Dir* is not constrained by the last line of *NewDirEntries*; to remedy this, one might change it to

```
all x: Dir + Dir' - d | x.entries' = x.entries
```

Alloy's tool can generate multiple instances of a formula. By exploting symmetries of the specification, it can ensure that only non-isomorphic instances are produced. For simulation, this is a boon, since one can examine every instance within a small scope to check that the behaviours or states are formed as intended.

The analysis is inherently intractable: addition of a single relational state component in a scope of 3 increases the state space by 3 orders of magnitude. Currently, the tool can handle a specification with 10 or 20 state components in a scope of 3 to 5. It seems unlikely that it will ever be possible to analyze a large, monolithic specification that has been written without analysis in mind. However, we have found that the analysis is powerful enough to allow small models to be constructed without such concerns, and that design problems are often readily decomposed into fairly independent models.

By its very nature, the analysis is not *complete*: a failure to find a counterexample does not prove a theorem correct. It may be possible to perform a static analysis that establishes a 'small model theorem'. If one can show that a formula has a model only if it has a model within a given scope, an analysis within that scope would consistitute a proof. Because Alloy is based on traditional logic and set notions, it may be possible to develop such a technique by applying known results from model theory.

## 3 Language Definition

Alloy is based on a tiny kernel language. The language as a whole is defined by translation to the kernel. Here the translation is given informally, but the kernel itself is presented with a formal semantics. Figure 2 gives an abstract syntax (on the left), a type system (in the middle) and a semantics (on the right). Most features of the kernel are standard, so we focus here on its novelties: the treatment of scalars as singleton sets, the encoding of sets as degenerate relations, and the dot operator used to form 'navigation expressions'.

### 3.1 Kernel Syntax

The kernel is strongly typed, and a formula is accompanied by declarations of the set and relation variables; we call the combination of a formula and its declarations a *problem*. Each declaration associates a type with a variable. There are three kinds of type:

- the *set* type $T$, denoting sets of atoms drawn from $T$;
- the *relation* type $S \rightarrow T$, denoting relations from $S$ to $T$;
- the *function* type $T \Rightarrow t$, denoting functions from atoms of $T$ to values of type $t$.

Types are constructed from primitive types that denote disjoint sets of atoms. We use upper case names for primitive types and lower case names for arbitrary types. So in the type $T \Rightarrow t$, the index type $T$ must be primitive, but $t$ may be a set type, relation type or another function type.

Functions correspond to predicates of arity greater than two, and generalize the OMT notion of 'qualified associations' [52]. The predicate *Rides* (*r,j,h*) that holds when jockey *j* rides horse *h* in race *r*, for example, might be declared as a function

$$rides : Race \Rightarrow Jockey \rightarrow Horse$$

and, for a given race *r*, the expression *rides*[*r*] would then denote a relation mapping jockeys to their

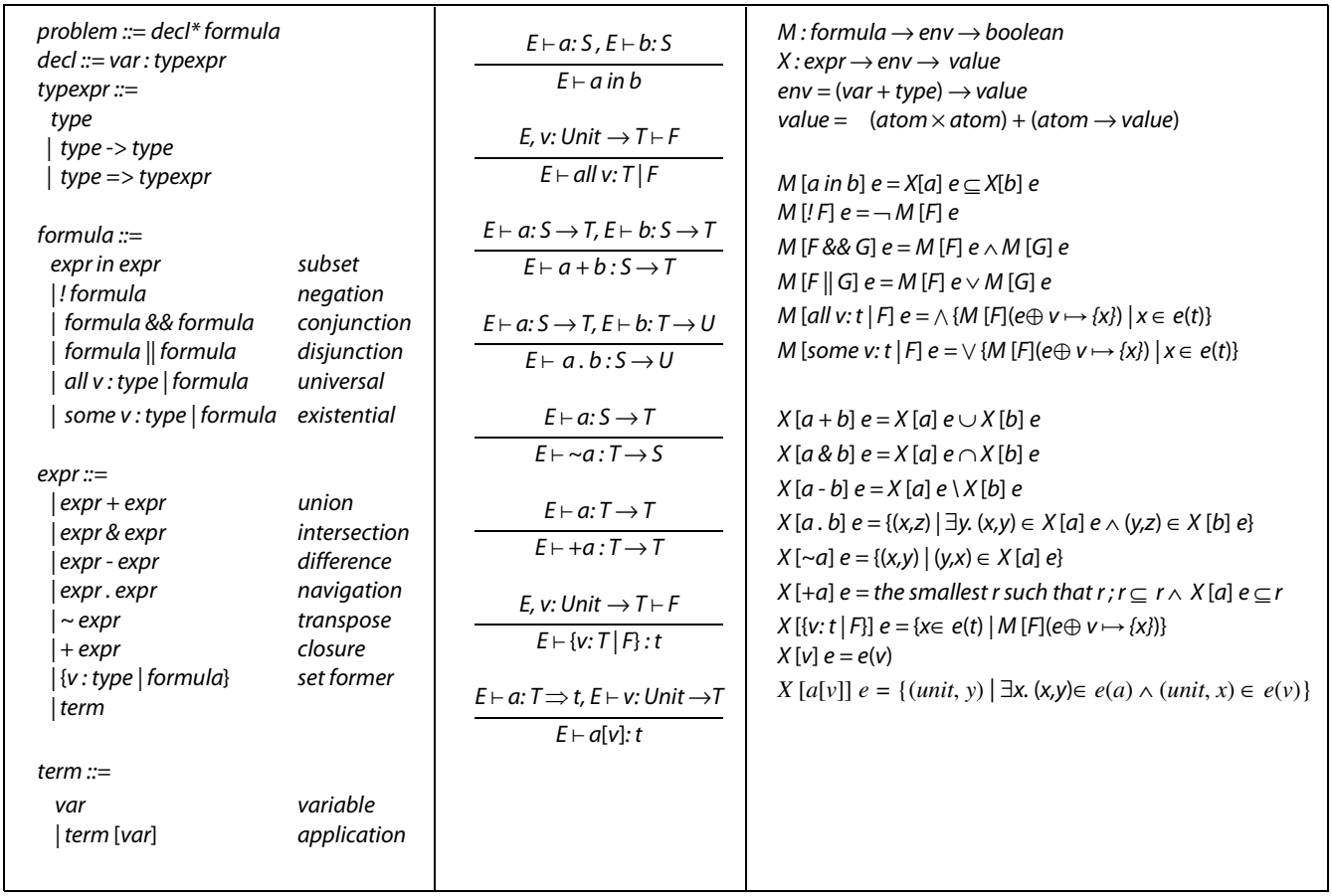



*Figure 2: Kernel syntax, type rules and semantics*

horses in that race. The current version of the Alloy tool does not support arbitrary functions, but just functions to relations; the above declaration would actually be written

*rides [Race] : Jockey -> Horse*

A function can be viewed as a curried form of a relation from a tuple; this function, for example, might have been declared as $Race \times Jockey \rightarrow Horse$. Functions, however, give a simpler expression syntax, since there are no tuples to construct and deconstruct. Of course, one can always introduce tuples explicitly in Alloy (eg, a type *Run* with projections to *Race* and *Jockey*) although they have no special syntactic support. In the file system example, *DirEntry* is such a tuple; instead, we might have declared a function

*contents [Object] : Name -> Object*

so that *contents*[$d$] gives a mapping for a directory $d$ from names to objects it contains.

Functions retain the binary flavour of the logic: they fit naturally into diagrams, and can accommodate multiplicity markings. In the full language, the question marks in

*rides [Race] : Jockey? -> Horse?*

for example, indicate that, in each race, a jockey rides at most one horse and vice versa. Finally, functions have an advantage over the introduction of tuples for tool implementation also, since

they can be used to represent skolem functions introduced to replace existential quantifiers.

There are no separate scalar types. To declare a scalar variable, we declare it to be a set

$v : T$

and add a constraint that makes the set a singleton:

$some\ x: T \mid x = v$

(exploiting the fact that, by definition, $x$ is bound to a singleton). This allows navigation expressions to be written uniformly, without the need to convert back and forth between scalars and sets, sidesteps the partial function problem, and simplifies the semantics (and its implementation).

Formulas have a conventional syntax. There is only one elementary formula, stating that one expression is a subset of another. In quantified formulas, the variable is declared to have primitive type, and is interpreted as being bound to singleton subsets of the type.

Expressions are formed using the standard set operators (union, intersection and difference), the unary relational operators (transpose and transitive closure), and the dot operator, used to form navigation expressions. The unary operators are prefixes, to make parsing easy.

Set comprehension has the standard form. Set and relation variables are expressions, but function variables, and functions in general, are not. Ensuring that functions can only be applied to variables guarantees that an expression involving a function is always well defined, since the function's argument will denote a singleton set.

### 3.2 Kernel Type System

We treat sets semantically as degenerate relations, viewing the set $\{e_1, e_2, \ldots\}$ as the relation $\{(unit, e_1), (unit, e_2), \ldots\}$ where *unit* is a special atom that is the sole member of a special type *Unit*. Unlike our treatment of scalars as singleton sets, this is just a trick that makes the semantics more uniform; users of Alloy will prefer to view sets in the traditional way. The type of a variable declared as *v: T* is thus represented as $Unit \rightarrow T$, although we shall sometimes write this type as $T$ for short.

The typing rules determine which problems are well-formed. The judgment $E \vdash a: t$ says that in the type environment $E$, expression $a$ has type $t$; the judgment $E \vdash F$ says that in environment $E$, the formula $F$ is well-typed. Obvious rules (eg, for conjunction) are omitted, as well as those (eg, for intersection) identical to rules given.

A problem is type checked in an initial environment that binds each variable to the type as declared. The environment is extended in the checking of quantified formulas and set comprehensions. For example, the rule for universal quantification says that the quantified formula is well-typed when its body is well-typed in the environment extended with the binding of the bound variable to its declared type.

The set operators can be applied to sets or relations; when + is applied to two sets, for example, the type variable $S$ in the typing rule will be bound to *Unit*. Likewise, the dot operator can be applied to sets or relations, in any combination that the typing allows. Note that the typing rules make clear where sets alone are legal: for bound variables, and the arguments of function applications.

### 3.3 Kernel Semantics

The meaning of the logic is defined by a standard denotational semantics. There are two meaning functions: $M$, which interprets a formula as true or false, and $X$, which interprets an expression as a value. Values are either binary relations over atoms, or functions from atoms to values. Interpretation is always in the context of an environment that binds variables and primitive types to values, so each meaning function takes both a syntactic object and an environment as arguments.

Each definition defines the meaning of an expression or formula in terms of its constituents. For example, the elementary formula *a in b* is true in the environment *e* when $X[a]e$, the relation denoted by *a* in *e,* is a subset of $X[b]e$, the relation denoted by *b* in *e*. The formula *all v*: $T \mid F$ is true in *e* when *F* is true in every environment $e \oplus v \mapsto \{x\}$ obtained by adding to *e* a binding of *v* to the singleton set containing *x*, where *x* is a member of the set denoted by the type *T* in *e* (and, due to the encoding of sets, will actually be a pair with *unit* as its first element). We assume that bound variables have been systematically renamed where necessary to avoid shadowing.

All operators have their standard interpretation. The dot operator is semantically just relational composition, but, because of the encoding of sets as relations, it does double duty. When *s* is a set and *r* is a relation, *s.r* denotes the image of *s* under *r*. This is the only form exploited in the full Alloy language; an expression such as *s.r* is usually called a *navigation expression*, since it suggests a navigation from the elements of *s* across the relation *r*. The kernel definition admits additional forms though: when *p* and *q* are relations, *p.q* is their join; and when *s* and *t* are sets, *s.~t* is their cross product.

The meaning of a problem is the collection of well-formed environments in which its formula evaluates to true. An environment is well-formed if: (1) it assigns values to the variables and basic types appearing in the problem's declarations, and (2) it is well-typed—namely that it assigns to each variable an appropriate value given the variable's type. For example, if a variable *v* has type $S \rightarrow T$ in an environment *e*, then $e(v)$, the value assigned to *v* in *e*, must be a relation from the set denoted by *S* to the set denoted by *T*.

The environments for which the formula is true are the *models* of the formula. To avoid that term's many overloadings, we shall call them *instances* instead. If a formula has at least one model, it is said to be *consistent*; when every well-formed environment is a model, the formula is *valid*. The negation of a valid formula is inconsistent, so to check an assertion, we look for a model of its negation; if one is found, it is a *counterexample*.

Since the kernel is undecidable, it is impossible to determine automatically whether a formula is valid or consistent. Alloy's analysis is limited to a finite *scope* that bounds the sizes of the carrier sets of the basic types. A model is *within a scope of k* if it assigns to each type a set consisting of no more than *k* elements. Clearly, if the analysis succeeds in finding a model to a formula, consistency is demonstrated. Failure to find a model within a given scope, however, does not prove that the formula is inconsistent (although in practice, for a large enough scope, it often strongly suggests it).

### 3.4 Full Language

The full Alloy language differs from the kernel in three respects: a more elaborate syntax for declarations, the addition of a variety of shorthands, and paragraph structuring.

### 3.5 Declarations: Mutability & Multiplicity

In contrast to the kernel, the full language does not allow explicit mention of types. A primitive type is generated for each domain (namely each set variable declared in the domain paragraph). This type has no name and thus cannot appear in the specification. Other set and relation variables are declared in terms of the domains, and from these declarations, types are inferred. For example,

```
domain {A, B}
```

declares two domains *A* and *B*, with types *TA* and *TB* say. A subsequent declaration

```
S : A
r : S -> B
```

introduces a set variable *S* of type *TA*, and a relation variable *r* of type $TA \longrightarrow TB$. The declarations express in addition the constraints

```
S in A
all x: TA | x.r in B
all y: TB | y.~r in S
```

bounding the set *S* and the domain and range of *r*.

Each declaration also introduces a primed version of the variable (with corresponding constraints), unless the variable is marked as *fixed*. As explained below (in Section 3.7), conditions may be included by name, in which case the condition namestands for the condition's body, or when the name is primed, a primed version of the body. Priming a formula affects only state variables (and not bound variables), and does not affect those declared as fixed.

Static markings express constraints on transitions. The declaration

```
S : static A
```

says that *S* is a static classification of elements of *A,* so that a member of *A* is always either in *S* or not in *S*), and gives the constraint

```
all x: A & A' | x & S = x & S'
```

The declaration

```
r : A -> static B
```

says that the set of elements of *B* associated with a given element of *A* does not change over its lifetime, and gives

```
all x: A & A' | x.r = x.r'
```

The declaration

```
r : static A -> B
```

says that the set of elements of A that map to a given element of B does not change over that element's lifetime, and gives

```
all y: B & B' | y.~r = y.~r'
```

A relation declaration with *static* on both sides would induce both the second and third constraint. Note that, even in this case, *r*'s value may still change.

Multiplicities are translated into constraints in the obvious way. The declaration

```
r : A -> B?
```

for example, gives the constraint that *r* is a partial function

```
all x: TA | sole x.r
```

where *sole e* is a formula (defined below) that is true when the expression *e* denotes a set of at most one element.

Whereas the kernel allows 'function' variables with any number of index types, the Alloy tool currently only supports a single index. One can declare an *indexed relation*

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

whose type will be $TI \Rightarrow TA \rightarrow TB$. Mutability and multiplicity constraints are generated as for regular relations, with an additional quantification over the index variable. For example,

```
r [I] : A -> B?
```

would give

```
all i: TI | all x: TA | sole x.r[i]
```

saying that each relation *r[i]* is a partial function.

### 3.6 Formula Shorthands

When no explicit bound is given in a quantified formula, a bound is inferred. Given a formula

```
all x | F
```

we first determine the type of the quantified variable *x*. If it is found to be *TA*, the type associated with a domain *A*, the formula is treated as short for

```
all x: A | F
```

An explicit bound may be given instead; the formula

```
all x: e | F
```

is short for

```
all x | x !in e || F
```

in which x is to be interpreted as bound only by its inferred type (and so the formula is actually not an Alloy formula). This approach to quantification has several advantages: it allows more succinct formulas; it naturally limits formulas about the state to constraints about atoms that exist; and it ensures the property of scope monotonicity (see Section 5.2).

A few operators missing from the kernel are trivially derived: implication, in terms of negation and disjunction, for example, and equality of expressions in terms of *in*. The symbol *!* can be used to negate operators; *a !in b*, for example, is short for *!(a in b)*. The additional quantifiers are defined as follows:

```
no x | F ≡ all x | !F
sole x | F ≡ some y | {x | F} in y
one x | F ≡ sole x | F && some x | F
```

Conventional shorthands for quantified formulas are included: one can quantify over several variables with the same quantifier, and bound the variable with an expression: *all x: e | F*, for example, is short for *all x | x !in e || F*. The type of the bound variable, required in the kernel syntax, is inferred from the body formula.

Quantifiers can be applied to expressions: the formula *Q e*, for a quantifier *Q* and an expression *e* is short for *Q v | v in e*. Thus *no e*, *some e*, *one e*, and *sole e* say that *e* has no elements, some elements, exactly one, and at most one. These forms allow the omission of set constants from the language.

The uniform treatment of scalars as sets, and the use of the navigation operator in place of function application, tends to make specifications shorter and easier to understand. It is not a panacea, however. Sometimes, when interpreting *in* as membership rather than subset, it comes as a surprise

that the formula will be true when the left hand side denotes an empty set. To say that nobody's wife is his sibling, for example, we might write

*no p | p.wife in p.siblings*

but unfortunately this implies that every person has a wife. What we mean instead is

*no p | some p.wife && p.wife in p.siblings*

To make such formulas more succinct, Alloy includes the 'strong' subset and equality operators */in* and /=, so this formula can be written

*no p | p.wife /in p.siblings*

### 3.7 Paragraph Structure

Formulas in Alloy are organized into *paragraphs*. The meaning of a paragraph includes not only the formula given explicitly, but also formulas imported from invariant and definition paragraphs, and the implicit formulas associated with declarations. The difference between a condition and an invariant is that an invariant is implicitly imported into all other paragraphs, but a condition is not (and may be explicitly parameterized). The Alloy tool has a switch to turn this importation on or off, and can check that operations preserve invariants without requiring explicit assertions.

Invariants and definitions do not differ semantically, but they are treated differently by the tool. A definition is expected to constrain only the defined variable, and to give it exactly one value for given values of the other variables. This not only allows additional checks to be performed, but can also be used to simplify the analysis. If a paragraph does not mention a particular variable, the analysis might omit a formula defining that variable, even though the definition mentions other variables that appear in the paragraph, on the grounds that the definition constrains only the defined variable This is sound so long as the definition is well formed. The current Alloy tool does not check well-formedness, but allows the user to indicate that definitions are to be 'believed' so that such omissions are enabled.

Condition and operation paragraphs can be explicitly 'invoked'. Following Z, the invocation *C'* is an invocation of the condition *C* with its variables primed. Unlike Z, Alloy allows conditions and operations to have arguments. Invocations with arguments are handled simply by replacing occurrences of the formal variable with the corresponding actual expression. Arguments, like state variables, can be declared in terms of sets; for operations, these may include sets in the post-state. The declaration *o: Object'!* from the *Create* operation (Figure 1), for example, constrains *o* to be a scalar drawn from the set of objects after execution.

## 4 Graphical Syntax

An example of Alloy's graphical syntax at the top of Figure 1. A diagram can only express state, and not operations, and only those constraints that are expressible in the domain and state paragraphs. The graphical elements correspond directly to the elements of the domain and state paragraphs.

Each set component is represented as a box. The set declaration *S : T m*, where *S* and *T* are sets and *m* is a multiplicity symbol, is represented as an arrow with a triangular head from a box marked *S m* to a box marked *T*. Domains are not specially marked; they are simply the sets with no outgoing triangular arrows. If a set is static, it has a vertical stripe down the left-hand side of the box; if it

is fixed, it has a stripe on the right too. Domains are by definition static, so no special marking is needed to indicate this, graphically or textually. Several sets may be connected to a single set with arrows that meet in a single arrowhead; this declares the sets to be disjoint. If the arrowhead is filled, the collection additionally forms a partition.

A relation *r* from *S* to *T* is represented as an arrow, with a smaller, open head, and labelled *r*, from the box labelled *S* to the box labelled *T*. Multiplicity symbols are written at the ends of the arrow. If the relation is source-static (that is, would have been declared textually as *static S -> T*), the tail of the arrow is marked with a small hatch; if it is target-static (declared textually as *S -> static T)* the head is marked.

Relations with identical declarations may share a single arrow, labelled with a comma-separated list of relation names. An indexed relation declared textually as

```
r [X] : S -> T
```

is represented by a relation arrow from *S* to *T* labelled *r*[*X*]. Similarly,

```
p (~q) : S -> T
```

is represented by a relation arrow from *S* to *T* labelled *p(~q)*.

### 4.1 Comparison to UML

This syntax is largely compatible with UML [53]. A comparison is tricky, since UML's meaning is defined in mostly implementation-oriented terms. UML diagrams do not have a corresponding textual syntax in OCL, so every UML model must include a diagram, in addition to any textual constraints.

Nevertheless, the basic notational differences are as follows:

- Alloy uses regular expression operators as multiplicity symbols instead of integer ranges. Aside from looking neater, the Alloy symbols sit more comfortably in text. Using UML symbols would change *r: A? -> B?* to *r: A [0..1] -> B [0..1]*, more than doubling the number of keystrokes.
- Alloy's treatment of scalars as singleton sets allows important singletons such as the root of a file system to be shown in the diagram.
- In Alloy's graphical syntax, a set may have at most one superset; UML has 'multiple inheritance'.
- Alloy's syntax for disjointness and exhaustiveness requires no textual annotations. InUML, since exhaustiveness is represented as a property of the superset, it is not possible to show more than one classification of a set. One cannot, for example, declare a set *Person*, with classifications into *Adult* and *Child*, and *Employed* and *Unemployed*. Oddly, UML makes no use of the visual distinction between sets that share a subtree and sets that do not. Catalysis [13], another UML variant influenced by Z, takes the same approach as Alloy.
- In place of a relation label *p(~q)*, UML would show the labels *p* and *q* at the arc ends. This seems to make diagrams harder to lay out, since it clutters the space around boxes.
- Mutability is indicated in UML with textual markings at the ends of associations. Classification is by default static. To loosen this implicit constraint, and allow a classification to be dynamic, a textual annotation is added to the diagram.
- Qualifiers in UML take the place of Alloy's indexed relations. Alloy does not have association classes.

· UML includes a variety of additional notions, mostly implementation oriented (such as navigability). Associations can be described as aggregations or compositions (discussed in Section 6.4 below).

# 5 Rationale

This section explains why Z was chosen as the basis for Alloy, but why Z alone was not deemed sufficient. Z has a simpler semantics than other formal specification languages, and, because of its relational nature, is well matched to object modelling. In addition to its technical superiority over UML's constraint language, it has also been far more widely used, and its power and limitations are better understood. Despite these benefits, though, Z has not been adopted in industry, and there is a widespread consensus that something different is needed. Alloy addresses three perceived deficiencies of Z: a lack of automatic tool support, incompatibility with object modelling idioms, and dependence on LaTeX.

## 5.1 Basing Alloy on Z

Z [58] was chosen as Alloy's starting point primarily because it has a simple and intuitive semantics that is well suited to object modelling. Its 'model-oriented' approach to specification, in which data structures are built from concrete mathematical structures, scales better than the 'algebraic' approach typified by Larch [21] and OBJ [19], in which data structures are characterized implicitly by their properties. It is also a more natural fit for data modelling, since relations are basic to Z but not easily described algebraically. Z has roots in the database work of Abrial [2], so its compatibility with the object modeling notations of OMT [52] and its successors, which are themselves based on semantic data models, is hardly surprising.

The underlying semantics of Z is almost trivially simple. Every data structure is represented using sets and tuples. A relation is a set of pairs; a function is a relation in which no two pairs share the same first element; a sequence is a function from natural numbers to the element type. A specification is constructed from logical formulas, and its meaning is given as sets of structures that satisfy formulas. An operation, for example, is a formula whose meaning is a set of transitions; each transition is a tuple consisting of components that represent the pre- and post-states.

VDM [42], although model-oriented, builds data structures from a library of algebraically-defined datatypes. Its semantics requires more than set theory: in particular, the notion of domains from denotational semantics, and a three-valued logic. Alloy's tool development has exploited the ability to reduce the language to a tiny kernel (Section 3.1); it is not clear this could be done so easily for a language such as VDM.

Given the current popularity of UML [53], one might ask why its constraint language, OCL [63], was not selected as the basis of Alloy. Most importantly, it seemed prudent to start with a language that was tried and tested, and would be a solid foundation for further development. Z has been used in large-scale industrial developments and widely taught; tools are available; and several collections of specifications have been published. There are, in contrast, no substantial publicly available examples of OCL models beyond the metamodel of UML itself, and few tools.

OCL was designed with much the same motivations as Alloy. It has its origins in Syntropy [9],

whose authors acknowledge the influence of Z. Like Alloy, OCL aims to be a formal language that is more easily used by non-mathematicians than Z. But while Alloy has simplified the underlying semantics of Z even further, OCL has complicated it, in order to accommodate a variety of notions from object-oriented programming. Articulating the essence of a system is for many practitioners much harder than writing code, but is not made easier by giving a code-like semantics to models. OCL's language features include parametric and subtype polymorphism, operator overloading, introspection, type casing and multiple inheritance – a challenging combination for a programming language, let alone a modelling notation (which one expects to be much smaller and simpler). A full semantics for OCL has yet to be worked out, and a variety of inconsistencies in its definition have been noted [5].

UML, whatever its defects, is at least an industry standard. On pragmatic grounds alone, perhaps it would make a better target for analysis than a new language. Although appealing, I believe this argument to be mistaken. First, although UML has been codified, in its use the language is still in flux. Notations and methods that are described as UML-compliant often deviate markedly from the standard; the Catalysis approach to UML [13], for example, uses a constraint language that has more in common with Alloy than OCL. Second, as mentioned above, OCL is not currently widely used, and is certainly less well known than Z. Third, this argument underestimates the difficulty of putting OCL into analyzable form, whether by translation, subsetting or by defining the semantics of the language as it is. The project of formalizing UML currently occupies a raft of researchers, and is likely to take several years. It seemed better to build tools now and apply them to real problems than to tackle (or wait for others to overcome) this formidable obstacle. Finally, in time, it may be that UML comes to be viewed unfavorably as a premature attempt at standardization, and that the cause of software development is better served by working on far simpler notations with more powerful tool support.

The relationship between Alloy and OCL is discussed further in Section 7.1.

### 5.2 Automatic Analysis

Z was not designed with automatic, semantic analysis in mind. To make the language more amenable to analysis, Alloy eliminates features that make analysis hard, provides syntactic support to allow the user to communicate with an analysis tool, and adjusts both the syntax and semantics to ease implementation of the analysis.

Analyses can be classified according to their semantic depth and degree of automation. Type checking is shallow but fully automatic; theorem proving is deep but typically requires guidance from the user. Alloy is designed for an analysis that is both deep and automatic.

Ideally, the language would be decidable. Unfortunately, the most elementary calculus that involves relations is undecidable – even Tarski's relational calculus, which has no quantifiers and is strictly less expressive than first order logic. Some compromise is thus inevitable. Alloy's analysis finds models of formulas: that is, assignments of values to variables for which the formula is true. When the formula is the negation of a theorem, its models are counterexamples; when the formula is a state invariant or operation, the models are samples (either instances of the state or transitions). The analysis is guaranteed to be sound, in the sense that a model returned will indeed be a model. There are therefore no false alarms, and samples are always legitimate (and demonstrate consis-

tency of the invariant or operation). On the other hand, when the analysis does not return a model, one cannot conclude that none exists. The validity of a theorem cannot be guaranteed, and an invariant or operation that appears to be inconsistent may in fact be consistent.

The analysis works by considering all potential models up to a given size, specified by a scope that limits the number of atoms in each primitive type. In practice, it seems that most interesting properties, whether samples or counterexamples, can be illustrated within a small scope. So the absence of a model within a scope gives some empirical evidence that none exists, becoming more credible as the scope is increased.

The analysis is explained elsewhere [31]. In short, the values of a relation are viewed as adjacency matrices. Each relational variable is encoded as a matrix of boolean variables whose dimensions are determined by the scope, and a boolean formula is constructed whose models correspond to models of the original formula. An off-the-shelf SAT solver is used to find solutions to the boolean formula.

This analysis method imposes certain restrictions on the language. Several of these might be lifted with progress in analysis technology, but it seems unlikely that the features omitted will not always incur some additional cost. There are three fundamental restrictions:

- All datatypes are first order, since a higher order value cannot be represented as an adjacency matrix. There are no relations that map relations, for example. Alloy does support functions from atoms to relations, but these are simply first-order relations in curried form, and can be represented by a multidimensional adjacency matrix.
- All quantifiers are first order. The analysis eliminates scalar quantifiers by forming explicit conjunctions and disjunctions over all possible values of the bound variable. Higher-order quantifiers cannot be eliminated in this way, because a relation or set has too many values. Skolemization can eliminate some higher-order quantifiers, but Alloy's design is based on the premise that *all* specifications should be analyzable.
- All types are uninterpreted; in Z jargon, all primitive types are 'given'. Interpreted types and their operators are not easy to encode in a boolean formula. A future release of Alloy will include integers with at least addition, subtraction and less than, but it seems unlikely that division and multiplication can be handled. The interchangeability of the atoms of an uninterpreted type can be exploited [35]; Alloy's tool adds symmetry-breaking predicates that can improve performance dramatically.

A second influence of analysis on the language arises from the user's need to communicate with the analysis tool. In Z, only convention distinguishes the role played by different schemas. The syntax does not indicate, for example, whether a schema represents an invariant or an operation. Z does not even separate theorems about the specification from the specification proper. Alloy takes from Larch [21] the idea of explicit assertions, formulas that are added purely for checkable redundancy. It also distinguishes invariants from definitions [41], exposing additional opportunities for checking: a definition, unlike an invariant, is expected to constrain a variable to have exactly one value (and is expected not to constrain the remaining variables).

A third influence arises from the architecture of the analysis tool itself. The semantics and translation scheme are simplified by ensuring that every expression and formula has a denotation (that is, a formal meaning). Alloy has this property, because of the omission of function application and the semantics of navigation, but Z does not, since a partial function may be applied outside its

domain. A more subtle issue is *scope monotonicity*. Alloy is designed so that whenever a formula has a model within a given scope, that model belongs also to any larger scope. This allows the tool to fix the size of the primitive types for a particular run, rather than considering all sizes smaller than the scope permits. To enable this, Alloy does not allow explicit reference to the sets that denote primitive types, so one cannot, for example, write a formula constraining a type to have exactly two atoms; in fact, types are implicit and are never named at all.

### 5.3 Set-Based Syntax

Z, although semantically founded on sets, gives better syntactic support to relations. Alloy includes some syntactic features (adapted from the syntax of informal notations) that make it easier to write models in terms of sets.

Relational operators are powerful and allow some constraints to be written far more succinctly than with quantifiers and set operators. The formula

```
no (parent & Id)
```

for example, says that no file system object is its own parent, by declaring to be empty the intersection of the *parent* relation and the identity relation that maps an object to itself. But formulas based on sets tend to be easier to read and write, and many would prefer

```
no p | p.parent = p
```

perhaps because its structure is closer to natural language.

Relational operators are rarely found in the informal notations. Some graphical notations allow a relation to be constrained to be a subset of another, or disjoint from it, but none of the textual constraint languages provides a union operator on relations. Two features of these informal notations support a style of specification in which sets rather than relations predominate, and have been incorporated into Alloy.

The first is the ability to give the domain and range of a relation as part of its declaration. In Alloy, the declarations

```
partition File, Dir : Object
parent : Object -> Dir?
```

introduce files and directories as disjoint subsets of the set of file system objects, and a relation that maps objects to their parent directories. The question mark in the second declaration indicates that each object has at most one parent.

Z does not allow a set to be declared and then used in the declaration of a relation in this way. One would have to add explicit constraints: that *File* and *Dir* partition *Object*, and that the range of *parent* is a subset of *Dir* (or that *parent* is an element of the set of partial functions from *Dir* to *Object*, which looks like an additional declaration). The most serious consequence of this limitation of Z's scoping rules is that it is not possible to transcribe an object model diagram directly into a sequence of declarations. It is especially unfortunate that properties of a relation that could otherwise have been expressed 'above the line' must be expressed below it. For example, the declaration

```
name : Dir -> Name!
```

says that every directory has exactly one name; in Z, one could declare the relation as functional, but not as total (since it is not total over *Object*).

The second feature is the 'navigation' syntax for relational image. Assume three additional dec-

larations

*Current : Dir!*
*Superuser: User!*
*owner : Object -> User!*

which introduce a current directory, a superuser, and a relation that maps each object to the user that owns it. The navigation expression *Current.owner* denotes the owner of the current directory; *Superuser.~owner* denotes the objects owned by the superuser; *Current.parent.owner* denotes the owner of the parent directory of the current directory; *Current.*parent* denotes the objects obtained by following *parent* zero or more times from the current directory, namely all its ancestors; *Current.*parent - Superuser.~owner* denotes the ancestors of the current directory not owned by the superuser; *(Current.*parent - Superuser.~owner).owner* denotes the owners of these objects; and so on.

A navigation operator already exists in Z: it is simply relational image. What makes navigation convenient in Alloy, however, is the type system, which treats scalars as singleton sets. No set braces are needed to lift a scalar to a set; the owner of the current directory is written *Current.owner* and not *{Current}.owner* . This is not a type coercion; there is no scalar type in Alloy. By providing only the navigation operator, and no function application, Alloy sidesteps the partial function problem: every expression denotes some set, there is no need for a notion of undefinedness or a null value, and formulas need not be guarded.

The 'multiplicity' syntax in Alloy is the same scheme used in all informal notations. There is a collection of multiplicity symbols; in Alloy, * (zero or more), + (for one or more), ! (exactly one), and ? (zero or one). Each relation may be marked with a symbol on each end; omission is equivalent to *. For symbols *m* and *n*, a relation declared as *S m -> T n* maps each atom in the set *S* to *n* atoms in the set *T*, and maps *m* atoms in *S* to each atom in *T*. This is more systematic than the scheme used in Z, and works well in diagrams too. Alloy goes a step further than current informal notations, and uses the same symbols to qualify sets: *S!* denotes a scalar, *S?* an 'option', and *S*+ a non-empty set.

## 5.4 Mutability

Z provides no explicit syntactic support for mutability constraints, beyond the delta/xi conventions. Alloy incorporates mutability notions from informal notations, and gives them a precise and abstract semantics.

In some informal object model notations, one can declare a classification of objects to be static (meaning that a given object cannot be reclassified during its lifetime), and can specify in what ways a relation can change. Formal notations such as Z tend not to support such notions explicitly. Alloy uses a single keyword – *static* – to mark a set as a static classification, and to mark a relation as immutable in one direction or another. This concept of relation mutability has a simple abstract semantics, but can also be used to describe an implementation.

In a file system model, for example, the relation *link* from an alias to the object it points to might be declared to be static (in the direction from alias to object):

*link: Alias -> static Object*

This would imply that which object an alias points to cannot be changed, although aliases can be created and destroyed:

*all a: Alias & Alias' | a.link = a.link'*

In an implementation, one might exploit this by making aliases immutable. But the declaration of *link* makes no such commitment; there is no requirement even that the relation be implemented as a field of an alias object.

In Z, every component declared in a state schema will have a primed counterpart in an operation. If the component is fixed, and one wants to use the same name in the pre- and post-state, the component must be given a global scope, compromising the modularity of the specification. Alloy therefore allows a set component to be declared as *fixed*, and the name of such a set cannot be primed.

### 5.5 Lexical Issues

A criticism of Z's dependence on LaTeX [45], and its use of special symbols and Greek letters, risks sounding philistine and mundane (and mean-spirited, since Z specifications are so visually attractive). But the issue is important in practice, and it is tricker to design a neat ASCII notation than one might imagine.

Standard formatting of Z requires typographic symbols not found even in commercial mathematical faces (such as Adobe's Mathematical Pi). Although a 'horizontal form' has been defined that does not need boxes, it is rarely used for an entire specification. Consequently, it is a burden to produce a Z specification that does not look amateurish without the aid of LaTeX. Many Z tools, including the indispensable FuZZ type checker, take only LaTeX source code as input.

The negative consequences of tying Z to LaTeX have not been sufficiently appreciated. Perhaps more than any other factor, this has limited the adoption of Z beyond circles in which LaTeX is the formatting tool of choice. A Z specifier who uses industry-standard tools (such as Word and PowerPoint) cannot easily include Z in documents. Specifications cannot be easily communicated by email, and they cannot be incorporated as annotations in code. Even in academia there is a serious problem: Z cannot be taught without also teaching the use of LaTeX, which can present bigger hurdles to novices than Z itself.

Alloy is a plain ASCII notation. Many operators use only a single keystroke: + and &, for example, stand for union and intersection. The rest use at most two, and are familiar from programming languages: &&, for example, stands for conjunction, and -> is used to construct relation types. There are only 19 reserved words.

## 6 Experience & Evaluation

Alloy has been in use for about two years. It has been applied in the Software Design Group at MIT in a variety of small projects, including: the analysis of a resource discovery system [43]; the design of an air-traffic control system component [65]; and (with Kevin Sullivan) the reformulation of some essential properties of Microsoft COM's query interface mechanism [37]. To allow a direct comparison of Alloy to UML, the core metamodel of UML [62] itself was translated from OCL to Alloy; the resulting Alloy specification [61] is shorter and simpler, and was shown to be consistent using Alloy's automatic analysis. Alloy is currently being evaluated in a variety of other applications: to security (describing the organization of domains and trust relationships); to software

architecture (describing legal configurations); and to the analysis of code [38].

Alloy has been taught in courses at about ten universities, usually along with two or three other notations and their analysis tools. Its graphical notation and a slight variant of the textual notation has been adopted in the undergraduate software engineering class at MIT [46].

No formal evaluation has been performed. Novices appear to have had little difficulty in learning the notation; a week generally suffices for an undergraduate with a basic grounding in discrete mathematics. The availability of an interactive tool, especially with visualization of instances (only recently added), seems to make a big difference. It gives immediate, concrete feedback, and lends to specification some of the emotional appeal of programming.

### 6.1 Language Design Faults

A number of deficiencies of the language have become apparent, a few from the comments of users, but mostly from the experience of the author and his research group in constructing the analysis tool and in developing Alloy models. Some are faults of language design that can be easily remedied: the omission of a *let* construct, for example, and minor differences in syntax between state declarations and bound variable declarations (the latter not supporting the *disjoint* keyword, for example).

Some deficiencies are omissions that were motivated by simplicity but appear to have been misguided. The two most significant are the requirement that indexed relations appearing in expressions be indexed by variables (which often calls for an additional spurious quantifier), which can be remedied with a slight change of semantics, and the omission of relational operators. When an expression of the form *e.*(p+q)* is needed, the lack of relational union forces definition of an additional relational variable. Although the set-based style works well for writing state invariants, it is less well suited to operations. In Z, the last two constraints of the *NewDirEntries* operation (Figure 1) would be written more succinctly as a single relational formula:

*entries' = entries + {d -> e}*

The notion of domains, and the inability to quantify directly over types (rather than domains), creates problems. It exacerbates the frame problem. In the same operation, the formula

*all x: Dir - d | x.entries' = x.entries*

does not constrain the value of *entries'* outside the domain *Dir*, and one must therefore add the frame condition

*Dir' = Dir*

(whose omission was detected by checking the assertion *EntriesCreated*). It also forces mention of post-state sets in argument declarations, which is unnatural (although perhaps logical). One can always cirumvent this problem by introducing a fixed domain representing the set, although this is not in the spirit of the language.

The need for frame conditions in operations is a constant annoyance. One might include a keyword to say that all unmentioned state components are to be regarded as unchanged. But the design of such a feature is fraught with difficulties. It exposes further distinctions between definitions and invariants, since one often wants to describe an operation in terms of defined components, or to leave the defined components to take on the value determined by the definition. The Catalysis syntax *new*D* for *D' - D* might be a nice addition. It may also make sense to treat an expression of the

form *a.r*, where *a* is a newly created atom and *r* is a right-static relation, as short for *a.r'*; in the *Create* operation of the file system example, *e.contents* would then refer to the contents of the new entry *e* after execution.

Alloy does not allow a set to be declared as a subset of two sets. Because the graphical notation is constrained by the syntax of the textual declarations (to allow direct transcription), this means that 'multiple inheritance' cannot be shown in the diagram. This is a nuisance, and could be remedied by allowing declarations to use arbitrary expressions on the right-hand side.

## 6.2 Extending Expressiveness

As progress is made in the analysis underlying Alloy, it should be possible to extend the notation with some desirable features. The most commonly requested are: integers (with at least addition, subtraction, comparison and set cardinality); sequence components (which can usually be simulated with an explicit ordering relation); recursively defined constraints; and sequencing of operations.

Sequencing of operations present more of a language design challenge than a tractability problem. Following Z, one could take the formula *op1 ; op2* to be short for

```
some s: State | op1(pre,s) && op2(s,post)
```

but this calls for a higher-order quantifier (since *State* is a placeholder here for the actual state components, which are relational). In most cases, such as checking an assertion of the form

```
op1 ; op2 -> property(pre,post)
```

the quantifier can be skolemized away. It might therefore be best to allow higher-order quantifiers in the language, but to regard specifications in which they cannot be eliminated as ill-formed.

## 6.3 Structuring

Alloy's paragraph structuring has proven to be easy for novices to grasp and a good basis for tool support. But it has serious deficiencies, from the point of view of both user and tool developer. First, despite the ability to separate constraints into distinct paragraphs (and to have multiple state paragraphs, a syntactic convenience permitted by the Alloy tool), the incremental construction of a model is not adequately reflected in its structure. Second, it is not possible to reuse specification fragments in different contexts: one cannot, for example, define a tree generically. Some of the most basic structures are the trickiest to describe (try to define an acyclic, undirected graph, for example), and a burden that could be lifted from the specifier with an appropriate library mechanism.

Handling primed variables adds a surprising amount of complexity to Alloy's tool. Partly this is due to the automatic importation of invariants, and the provision of fixed components; the tool determines, for example, whether an assertion mentions post-state variables, and if so, introduces the appropriate invariants. (Incidentally, one motivation for this mechanism was to eliminate a complication of explicit imports: the need to resolve multiple declarations of the same component.) But it arises also in the need to apply renamings, when obtaining an invariant in primed form, and in the ad hoc nature of the code that organizes the results of an analysis into pre- and post-state components.

The notion of decoration, inherent in the schema calculus, lies at the root of these problems. It

would be more intuitive to regard an operation as a constraint over two structures, *pre* and *post* say, of the same type. Flattening these into a single structure, with elements of the latter marked by primes, brings succinctness and a certain simplicity to the expression of operations, but it makes it much harder to organize a tool systematically, and to provide clean composition mechanisms.

### 6.4 Attributes and Aggregation

Like Z and most other formal notations, Alloy makes no distinction between attributes and relations. Most informal notations, such as UML, use a different syntax for attributes, and sometimes even a different semantics; OCL, for example, uses null values for attributes but not for relations that map an object to zero or one objects.

Whether attributes deserve some special treatment is unclear. Certainly they matter when describing the details of a data model, but for the kind of structural modelling to which Alloy has been applied, where the focus on global structural properties, it is usually better either to omit attributes (because they are not significant), or promote them to the same level as other relations. In the file system example (Figure 1), it seems unnatural to treat the *name* relation as any less significant than the others; many complications in such a system are about naming, after all. For the same reason, the lack of integers in Alloy is not as serious as one might expect, since constraints involving integers are usually of secondary importance.

Aggregation is a thorny issue, and a subject of endless debate (and confusion). It may be useful at the implementation level, but it has no clear application in abstract models (and is therefore shunned by many authors, eg [13, 15]). Moreover, most attempts to give a semantics to aggregation result in a notion of immutability or lack of sharing which are easily expressed directly.

## 7 Related Work

### 7.1 UML's Object Constraint Language

The Object Constraint Language [63] of UML was designed with many of the same goals as Alloy. In some respects it is more expressive than Alloy: it has integer and string datatypes, and sequences. It does not have transitive closure, however, and the attempt to form closures using operations (as used repeatedly in the UML metamodel [62]) is not well founded. Indeed, the reason for including closure is that it is known not to be expressible in first-order logic.

Giving OCL a semantics is likely to be challenging, because of its rather elaborate type system (which has been shown to admit Russell's paradox [5]), implicit coercions, inheritance, and iteration construct. Progress has been made for a subset [23, 51], but it remains to be seen whether the entire language can be defined cleanly without significant change.

The syntax of OCL has been criticized – eg, by the authors of Catalysis [13], who propose a variant similar to Alloy – for being hard to read and write. OCL's expressions are stacked in the style of Smalltalk, which makes it hard to see the scope of quantified variables. Navigations are applied to atoms and not sets of atoms, although there is a *collect* operation that maps a function over a set. Attributes, which are treated no differently from relations in Alloy, are partial functions in OCL, and result in expressions with undefined value.

As an example of an OCL constraint, suppose we want to say that nobody gives a dog the name of a child or friend. In Alloy, we might write

```
all p: Person | all d: p.pets & Dog | d.name !in (p.children + p.friends).name
```

which can be read 'for all persons *p*, and for all *d* that are pets of *p* and dogs, *d*'s name is not in the set of names of *p*'s children and friends', or, making even more use of the ability to navigate from sets,

```
all p: Person | no (p.pets & Dog).name & (p.children + p.friends).name
```

In OCL, this statement might be written:

```
Person
self.pets->select (d | d.oclKindOf(Dog))
    ->forall(d | not (self.children->union(self.friends)->collect(q | q.name))->includes(d.name)
```

What makes this harder to read is the use of unary rather than binary operators, thus breaking the symmetry of terms such as *p.children + p.friends*; the distinction between types and sets, which forces the use of the special *oclKindOf* operator, and prevents use of the type *Dog* as an expression; and the definition of navigation in terms of scalars rather than sets, which adds two unnecessary quantifiers, *forall* and *collect*.

(This example is taken from a constraint in the metamodel of UML [62, Foundation Package: Core, Classifier, Rule 4] that reads: 'The name of an attribute [of a Classifier] may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier'. Further comparisons of OCL and Alloy formulations of UML metamodel constraints are given in [61]).

Alloy's notion of static relations formalizes the 'frozen' annotation of UML (which like many diagrammatic features in UML, seems to be available only in graphical form and not in OCL). Attention to mutability properties during modelling is well rewarded; not only does it often expose subtle problems, but it also provides a grounding for investigating where immutable datatypes may be used in the implementation. This is why mutability is treated as vital in Alloy, and not as a refinement.

## 7.2 Z Variants

Developers of several tools for Z, such as ZTC [64], have found the need for an ASCII version of the syntax. These tend to follow the layout of Z closely, which makes them sit more comfortably in an environment that also supports standard typeset Z, obtained from LaTeX input. A bijection from *A* to *B*, for example, might be denoted *A* >->> *B*, compared to *A!* -> *B!* in Alloy. In my experience (with NP, a predecessor of Alloy [30,34]), the desire to show the more attractive, typeset version of the notation in publications undermines the ASCII form. Typing even a few extra keystrokes is also surprisingly annoying; this is why, for example, the relation type constructor is -> in Alloy, and not <-> (as in NP), which is more compatible with Z.

Richard Botting has invented an ASCII syntax for mathematics, and has transcribed into it a variety of mathematical theories relevant to software [6].

Most theorem provers for Z (such as Z/EVES [11]) extend the language to distinguish theorems from the specification proper. The idea of including redundant assertions to a specification goes back at least to Larch [21]. Many specification languages impose more structure than Z: VDM, for example, distinguishes invariants (and splits operations into pre- and post-conditions). The difference between definitions and invariants (discussed at length in [41]) is recognized by fewer lan-

guages. Oddly, the notion is present in UML's graphical notation (in derived relations), but not in OCL.

### 7.3 Semantic Data Models

The fundamental idea of object modelling—namely constructing models of state in which object identity plays a central role—originates in the semantic data models developed in the context of databases [29,49]. One of the earliest and best known is Chen's entity-relationship model [10]; it was followed by more elaborate and sophisticated models, such as SDM [25] and DAPLEX [56]. Abiteboul and Hull formalized many of the notions present in these models in their IFO model [1]. These models do not include a full first-order logic, so they are not as expressive as Alloy or OCL.

The insight that a semantic data model is a better starting point for object-oriented development than a model in which objects have methods and fields is due to OMT [52]. Alloy's attempt to give an entirely abstract, implementation-free semantics to object models is in line with OMT's advocacy of 'analysis models' that capture problem properties without preempting later development decisions.

### 7.4 Formalizing Object Models

Several recent object-oriented methods have used ideas from formal specification, most often Z, to bring rigour to and extend the expressiveness of object models. Like Alloy, Fusion [8] views the object model as an invariant on the state, and allows transitions to be expressed as global operations. It does not have a formal constraint language. Syntropy [9] introduced the notion of navigation expressions, adopted later by UML, and although not formal, addresses carefully many issues treated superficially in UML (such as the meaning of aggregation). Catalysis [13] embodies a variant of UML, with a cleaner syntax and a semantics based on sets, similar to Alloy.

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 analyzable model. Larch is a popular target because its Shared Language provides a mathematical foundation that is not biased towards any style of specification structuring [4,5, 23]. Bickford and Guaspari's translation [5] 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 a closer fit than the operators of an algebraic language, and allow a more direct translation. UML classes are usually represented in Z with schema types [16, 17], following the object-oriented style of [22]. More recently, work has begun on a formalization of UML in which semantic functions are defined in Z [14].

### 7.5 Visual Logic

Visual formalisms for set theory and logic have a long history. Most well known are Venn diagrams and Euler circles. In a Venn diagram, a set is shown as a closed contour, the region inside the contour representing the contents of the set. All set terms are shown by overlapping the contours in

every combination; one then indicates which regions are empty and which are non-empty. When there are than a few sets, a Venn diagram is too hard to draw. Euler's scheme overcomes this problem by allowing overlappings to be omitted, and interpreting missing regions as empty sets. Harel's higraphs [26] are Euler circles superimposed with arrows between sets to indicate binary relations; his statecharts are higraphs in which the arrows represent transitions. The mathematician Charles Pierce developed a scheme for expressing logic diagrammatically. This work has been brought to the attention of computer scientists by Sowa, whose conceptual graphs [57] are based on Pierce's diagrams. (Many of these diagrams are described in [24].)

Given the advantages of graphical notations, it makes sense to try and express object model constraints visually. An extension of Pierce's diagrams have been developed for this purpose [18]. How well these approaches scale remains to be seen. Even drawing non-trivial object models as higraphs is difficult, and it may be that the 'classification hierarchy plus relations' style that has become almost universal represents the most that can be expressed conveniently in diagrammatic form.

## 7.6 Scalars and Partial Functions

The idea of treating a scalar as a singleton set appears in Quine's set theory [50] and in Tarski and Givant's relational reconstruction of set theory [59]. Hehner proposed the idea of a *bunch*, a data structure just like a set but of the same type as its elements, as a practical device for simplifying notation [28]. In bunch theory, however, a bunch is used to represent non-deterministic choice of a scalar value; functions take scalars as arguments, and the meaning of application to a bunch is obtained by applying the function pointwise over the bunch elements [7]. A set that is actually used as a value in the computation would *not* be represented by a bunch.

Different specification languages have addressed the problem of partial functions in different ways. Roughly speaking, the common approaches are: to extend types with special values (such as 'null') to record the result of an out-of-domain application [55]; to assign a third logical value to the smallest subformula containing the bad application, and compute the value of the formula as a whole in an extended, three-valued logic [42]; to regard all functions as total, and represent a partial function with an explicit domain outside of which the value of the function is unknown [21,20]; to treat the result of a bad application as well defined, but of unknown value [3,60]; and to give the value *false* to the smallest subformula containing the bad application [48,44].

Each approach has its merits, and none is ideal. Alloy's approach likewise is no panacea, but works well for the task at hand. Its context differs from most of the existing approaches in two ways. First, Alloy has limited expressiveness; the lack of function values, in particular, makes the elimination of scalars viable. Second, where most languages have been designed for syntactic proof, and the choice of partial-function scheme has been based on which axioms and proof rules hold, Alloy is designed for semantic analysis. For this, it helps if every expression and formula has a known denotation (for a given binding of free variables) that ideally can be computed compositionally.

Alloy's scheme, in addition to this property, requires no special values or non-standard logic. It also minimizes the need for guards [54]. The statement that some object has the root object as its parent

*some o: Object | o.parent = Root*

would require a guard in Z

```
some o: Object | o in dom parent && o.parent = Root
```

since the subformula *o.parent = Root* may be true for some value of *o* outside the domain of *parent*.

The last scheme mentioned above (assigning *false* to elementary formulas containing bad applications), used in Alloy's predecessor NP, has all these advantages too, but it has a nasty consequence. Because elementary formulas are given special treatment, *e1 != e2* is not in general equivalent to *(!e1 = e2)*. The 3-valued logic approach does not have this problem, since the negation of the third logical value is itself.

## 8 Conclusions

Why yet another modelling language? A series of observations motivated the development of Alloy. First, that first-order logic with sets and relations goes a long way – a large class of structural models can be described in Z without higher-order features, and can thus be analyzed efficiently. Second, that the object models advocated for problem modelling and analysis (by OMT, Syntropy, Fusion, Catalysis, UML, etc) are at heart semantic data models, and are best given a relational semantics. Third, that these models and their associated textual annotation languages offer some convenient syntactic features missing from Z.

In its details, Alloy is not novel; indeed its aim is to combine familiar and well-tested ideas from existing notations. Alloy's contribution is to show how a practical modelling notation can be assembled from only a handful of features. A few auxiliary notions allow these features to be fitted together smoothly. The implicit typing of domains, for example, allows the hierarchies of object models to be given a straightforward Z-style semantics, without the need to declare given types. The treatment of scalars as singleton sets makes navigation expressions uniform, and avoids the complications of undefinedness or special null values. Alloy's kernel can be viewed as an attempt to capture the essence of object modelling, and gives a precise semantics to the full notation in a simple and straightforward way.

Concrete syntax has been given more attention in Alloy's design than is currently fashionable (at least in academic circles). No unfamiliar-looking operators are used; punctuation is minimal; and keywords are used sparingly. When in doubt, a construct has been omitted: Alloy has no attributes, no scalars, no set constants, and no tuples.

Despite its minimality – or perhaps because of it – Alloy has turned out to be a useful notation. Small notations are often easier to read and write, and always easier to implement. Much of Alloy's benefit, however, comes from its tool. The ability to experiment with a model, generate samples, and check properties, changes the very nature of modelling. It gives a sense of concrete progress, and exposes errors more rapidly and ruthlessly than an expert reader. Fortunately, in Alloy's design, the the demands of tool developer and specifier have been not only compatible but symbiotic.

Since the writing of this paper, a new version of Alloy [36] has been developed that overcomes many of the flaws of the version described here.

## Acknowledgments

This paper benefited greatly from discussions about Alloy with Anthony Hall, Peter Henderson, Tony Hoare, Michael Jackson, Butler Lampson and Barbara Liskov, and from the comments of the

anonymous reviewers. Ian Schechter, Ilya Shlyakhter and Manu Sridharan helped implement the Alloy tool. Many users of Alloy, especially Mandana Vaziri and Sarfraz Khurshid, contributed to my understanding of the language.

This research was funded by the National Science Foundation (under grant CCR-9523972), by the MIT Center for Innovation in Product Development (under NSF Cooperative Agreement Number EEC-9529140), by NASA Ames (under grant NAG2-1338), and by an endowment from Douglas and Pat Ross. The views expressed herein are the author's and are not endorsed by the sponsors.

## 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] J.R. Abrial. Data Semantics. In J.W. Klimbie and K.L. Koffeman (eds.), *Data Base Management*. North Holland, 1974.

[3] R. Arthan. Undefinedness in Z: Issues for specification and proof. *CADE-13 Workshop on Mechanization of Partial Functions*, Rutgers University, New Brunswick, NJ, July 1996.

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

[5] Mark Bickford and David Guaspari. *Lightweight Analysis of UML*. TM-98-0036, Odyssey Research Associates, Ithaca, NY, November 1998.

[6] Richard Botting. *Maths in Ascii*. Poster. Joint Meeting of Southern California Chapter of Mathematical Association of America and Society for Industrial and Applied Mathematics (SIAM). Spring 1992. Text available at: http://www.csci.csusb.edu/dick/papers/rjb92b.discrete.

[7] A. Bunkenburg and J. M. Morris, A Theory of Bunches, Acta Informatica, to appear.

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

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

[10] 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.

[11] Dan Craigen, Irwin Meisels, and Mark Saaltink. Analysing Z Specifications with Z/EVES. In *Industrial-Strength Formal Methods in Practice*, J.P. Bowen and M.G. Hinchey (eds.), Springer Verlag, September 1999.

[12] Craig A. Damon, Ralph Melton, Robert J. Allen, Elizabeth Bigelow, James M. Ivers and David Garlan. *Formalizing a Specification for Analysis: The HLA Ownership Properties*. Technical Report CMU-CS-99-126, School of Computer Science. Carnegie Mellon University, Pittsburgh, PA, April 1999.

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

[14] 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.

[15] Martin Fowler. *Analysis Patterns: Reusable Object Models.* Addison Wesley, 1997.

[16] 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.

[17] 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.

[18] J. Gil, J. Howse, and S. Kent. Constraint Diagrams: A Step Beyond UML. In *Proceedings of Tools USA'99*. IEEE Computer Society Press, December 1999.

[19] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, ed., *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, eds., Addison Wesley, 1985, pages 391–420.

[20] David Gries and Fred B. Schneider. *Avoiding the Undefined by Underspecification.* In Jan van Leeuwen (ed.), *Computer Science Today: Recent Trends and Developments*, pages 366–373. Volume 1000, Lecture Notes in Computer Science, Springer-Verlag, NY, 1995.

[21] 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.

[22] 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.

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

[24] Eric M. Hammer. *Logic and Visual Information*. Center for the Study of Language and Information, Stanford University, Stanford, CA, 1995.

[25] 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.

[26] David Harel. On visual formalisms. *Communications of the ACM*, Vol. 31, No. 5, pp. 514-530, 1988.

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

[28] Eric C.R. Hehner. Bunch Theory: A Simple Set Theory For Computer Science. *Information Processing Letters*, Vol. 12, No. 1, February 1981, pp. 26–30.

[29] R. Hull and R. King. Semantic Data Models. *ACM Computing Surveys,* Vol. 20, No. 3, 1987, pp. 153–189.

[30] Daniel Jackson. Nitpick: A Checkable Specification language. *Proc. First ACM SIGSOFT Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996, pp. 60–69.

[31] Daniel Jackson. Automatic Analysis of Architectural Style. Unpublished manuscript. August 1997.

[32] Daniel Jackson and John Chapin. Redesigning Air-traffic Control: An Exercise in Software Design. *IEEE Software,* May/June 2000.

[33] Daniel Jackson and Craig Damon. *Semi-executable Specifications.* Technical report CMU-CS-95-216, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1995.

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

[35] Daniel Jackson, Somesh Jha and Craig A. Damon. Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications. *ACM Trans. Programming Languages and Systems*, Vol. 20, No. 2, March 1998, pp. 302–343.

[36] Daniel Jackson, Ilya Shlyakhter and Manu Sridharan. A Micromodularity Mechanism. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference* (FSE/ESEC '01), Vienna, September 2001.

[37] Daniel Jackson and Kevin Sullivan. COM Revisited: Tool Assisted Modelling and Analysis of Software Structures. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*. San Diego, November 2000.

[38] Daniel Jackson and Mandana Vaziri. Finding Bugs with a Constraint Solver. *International Symposium on Software Testing and Analysis (ISSTA'2000)*, Portland, OR, August 2000.

[39] Daniel Jackson, Yuchung Ng and Jeannette Wing. A Nitpick Analysis of IPv6. *Formal Aspects of Computing*.

[40] Daniel Jackson, Ian Schechter and Ilya Shlyakhter. Alcoa: the Alloy Constraint Analyzer. *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.

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

[42] Cliff Jones. *Systematic Software Development Using VDM*. Second edition, Prentice Hall, 1990.

[43] Sarfraz Khurshid and Daniel Jackson. Exploring the Design of an Intentional Naming Scheme with an Automatic Constraint Analyzer. *Proc. Automated Software Engineering*, Grenoble, France, September 2000.

[44] C.P.J. Koymans and G.R. Renardel de Lavalette. The logic MPLw. *Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J.A. Bergstra (eds.), LNCS 394, Springer Verlag, 1989, pp. 247–282.

[45] Leslie Lamport. *LaTeX : a document preparation system*. Addison-Wesley, 1986.

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

[47] Daniel Le Metayer. Software Architecture Styles as Graph Grammars. *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes, Vol. 21, No. 6, ACM Press, October 1996, pp. 3–14.

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

[49] J. Peckham and F. Maryanski. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Vol. 19, No. 3, 1988, pp. 201–260.

[50] W.V.O. Quine. New Foundations for Mathematical Logic. *American Mathematical Monthly*, Vol. 44, 1937, pp. 70–80.

[51] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and

Jos Warmer, editors, Advances in Object Modelling with the OCL, pages 38-63. Springer, Berlin, LNCS, 2001.

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

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

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

[55] Dana S. Scott. Existence and Description in Formal Logic. In *Bertran Russell, Philosopher of the Century*, R. Schoenmann, ed., Allen and Unwin, 1967, pp. 181–200.

[56] 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.

[57] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison Wesley, Reading, MA, 1984.

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

[59] Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*. American Mathematical Society Colloquium Publications, Volume 41, 1987.

[60] Sam H. Valentine. Inconsistency and Undefinedness in Z – A Practical Guide. *11th International Conference of Z Users (ZUM'98)*, Berlin, Germany, 1998.

[61] Mandana Vaziri and Daniel Jackson. *Some Shortcomings of OCL, the Object Constraint Language of UML*. Response to Object Management Group's Request for Information on UML 2.0, December 1999.Available at http://sdg.lcs.mit.edu/~dnj/publications.

[62] UML Partners. *UML Semantics*. Version 1.1, September 1997. Available at http://www.rational.com.

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

[64] Xiaoping Jia. *ZTC: A Type Checker for Z Notation*. User's Guide, Version 2.03. Division of Software Engineering School of Computer Science, Telecommunication, and Information Systems, DePaul University, Chicago, IL, August 1998.

[65] David Zhang. *Design of the Collaborative Arrival Planner using Object Modeling*. MEng. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2000.

## Appendix: Alloy Grammar

Vertical bar denotes choice; star denotes zero or more repetitions; angle brackets indicate optional phrases; *x,* indicates a comma-separated lists of *x*'s. *STAR*, *BAR* and *PRIME* are terminals representing an asterisk, vertical bar and prime mark respectively.

*model* ::= **model** *model-name* **{ domain {***domdecl, **}** para**}**
*para* ::= **state** ⟨*name*⟩ **{** *compdecl**}**
| **inv** ⟨*name*⟩ **{** *formula** **}**
| **def** *comp* **{** *formula** **}**
| **cond** *name* ⟨*arglist*⟩ **{** *formula** **}**
| **assert** ⟨*name*⟩ **{** *formula***}**
| **op** *name* ⟨*arglist*⟩ **{** *formula***}**
*domdecl* ::= ⟨**fixed**⟩ *set*
*compdecl* ::= *setdecl* | *reldecl*
*setdecl* ::= ⟨**disjoint** | **partition**⟩ *set*, **:** ⟨**fixed** | **static**⟩ *set mult*
*reldecl* ::= *relx* ⟨**(~** *relx***)**⟩ , **:** ⟨**static**⟩ *set mult* **->** ⟨**static**⟩ *set mult*
*relx* ::= *rel* | *rel* **[***set***]**
*mult* ::= **?** | **!** | **+**
*arglist* ::= **(** *argdecl,* **)**
*argdecl* ::= *arg,* **:** *set mult*
*formula* ::= *negate formula*
| *formula logic-op formula*
| *quantifier var-decl* , ***BAR*** *formula*
| *expr comp-op expr*
| *quantifier expr*
| **(** *formula* **)**
| *name* ⟨**(** *expr* , **)**⟩
*var-decl* ::= *var* , ⟨**:** *expr*⟩
*logic-op* ::= **&&** | **||** | **->** | **<->**
*negate* ***::=*** **not** | **!**
*comp-op* ::= **in** | **=** | *negate* **in** | *negate* **=** | **!=** | **!in**
*quantifier* ::= **all** | **some** | **no** | **sole** | **one**
*expr* ::= *var* | *arg* | *set* | *expr expr-op expr* | *expr* **.** *qualifier* | **{** *var-decl* ***BAR*** *formula* **}** | **(** *expr* **)**
*expr-op* ::= **+** | **-** | **&**
*qualifier* ::= *rel* | *rel* **[** *var* **]** | **~** *qualifier* | **+** *qualifier* | ***STAR*** *qualifier*
*arg ::= id*
*var ::= id*
*name ::= id*
*set ::= id* | *id* ***PRIME***
*rel ::= id* | *id* ***PRIME***