A Comparison of Object Modelling Notations: Alloy, UML and Z

Daniel Jackson
MIT Lab for Computer Science
August 11, 1999

Abstract

An example of an object model is given in full in three languages: Alloy, a new notation; Z, a formal specification language; and UML, a modelling notation popular in industry. Basic features of Alloy are explained informally, and briefly justified by comparison of the Alloy version to the UML and Z versions.

Introduction

Our example is a description of relationships amongst family members. It was chosen because the domain is familiar to readers, and thus has a rather contrived flavour. A real object model prepared in the development of a genealogical database would have rather different concerns, such as lack of information about relationships.

We show the example in each of the three notations: Alloy, Z [Spi92] and UML [RJB99]. Although we hope that some of the good and bad points of each notation will be evident from the examples and our discussion, this paper is not intended to be a complete description of Alloy, nor a justification of its design. A separate report [Jac99] gives a syntax and semantics for Alloy and explains its rationale in detail.

1 Alloy Version

The Alloy model has two parts. The graphical part (Figure 1a) is equivalent to the declarations of the textual part (Figure 1b): that is, to the paragraphs marked *domain* and *state*.

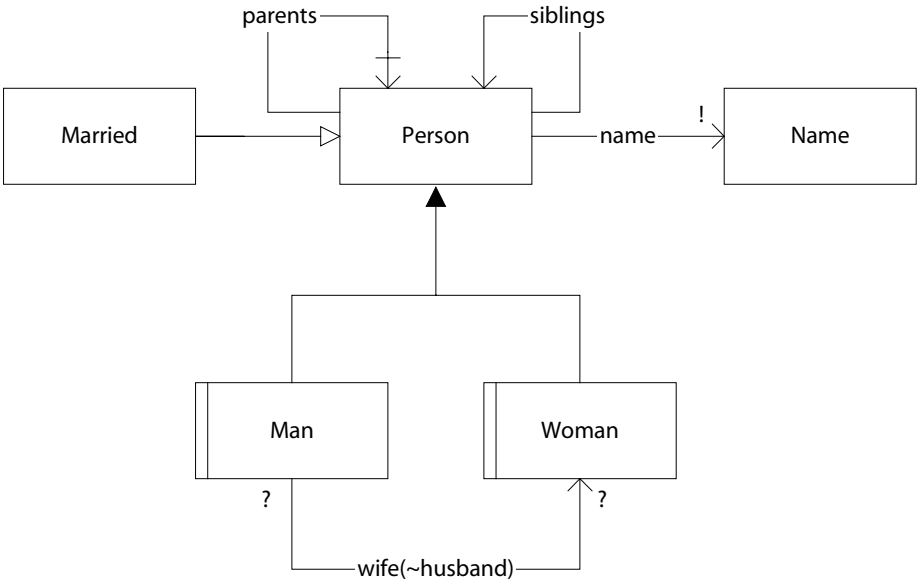

Figure 1a: Partial object model for a family tree

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

Figure 1b: *Alloy model for a family tree*

Sets, Domains, Types and Relations

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

and *Woman* partition *Person*: every *Person* is either a *Man* or a *Woman*. An arrow with an open head denotes a relation: *name*, for example, is a relation that maps *Person* to *Name*.

Sets without supersets, such as *Name* and *Person*, are called *domains*, and are implicitly disjoint. A type *ty* (*D*) is implicitly associated with each domain *D*. The expression (*Man* + *Woman*), denoting the union of the two sets *Man* and *Woman*, is legal because both *Man* and *Woman* have the type *ty* (*Person*); but the expression (*Man* + *Name*) is not, because *Man* has the type *ty* (*Person*) and *Name* has the type *ty* (*Name*).

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

Mutability Constraints

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

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

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

Textual Declarations

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

```
S:T
```

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

$$r : S \rightarrow T$$

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

Organization of Constraints

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

Navigation and Quantification

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

Expressions

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

Even variables take on set values, although these are implicitly constrained always to be singleton sets. The term *e1 in e2* is true when the set denoted by *e1* is a subset of that denoted by *e2*, so the term (*p in Man & Married*) has the same meaning it would have were *p* to denote an element and *in* to denote set membership. Encoding scalars as sets is a useful trick that allows functions and relations to be treated uniformly (since there is no function application distinct from relational image), and resolves the problem of partial functions applied outside their domain in a simple and pragmatic fashion. If *f* is a function that does

not map *x*, the expression *x.f* denotes the empty set, so if *y* is a scalar (and thus a singleton set), *x.f = y* is false.

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

```
e1 /in e2
```

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

Qualifiers and Relational Operators

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

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

for example, *p.+parents* is the image of *p* under the transitive closure of the *parents* relation—the set of objects obtained by following parents once, then again, and again, and so on until no further objects are added—in other words, *p*'s ancestors. The constraint thus states that no person is his or her own ancestor.

2 Z Version

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

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

[PERSON, NAME]

FamilyDecls

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

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

FamilyInv1

FamilyDecls

$siblings = \{a, b : PERSON \mid parents\ (\{a\}) = parents\ (\{b\}) \bullet a \mapsto b\}$
disjoint $\langle wife, siblings \rangle$
$parents \rhd Man \in PERSON \rightarrow\!\!\!\!\!\!\cdot\ \ PERSON$
$parents \rhd Woman \in PERSON \rightarrow\!\!\!\!\!\!\cdot\ \ PERSON$
disjoint $\langle parents^+, \mathrm{id}\ PERSON \rangle$
disjoint $\langle name\ ;\ name\sim\ ,\ parents\ ;\ parents\sim \rangle$

FamilyInv2

FamilyDecls

$\forall\ a, b : Person \bullet a \mapsto b \in siblings \Leftrightarrow parents\ (\{a\}) = parents\ (\{b\})$
$\nexists\ p : Person \bullet p \in \mathrm{dom}\ wife \wedge wife(p) \in siblings\ (\{p\})$
$\forall\ p : Person \bullet \#(parents\ (\{p\}) \cap Man) \leq 1$
$\forall\ p : Person \bullet \#(parents\ (\{p\}) \cap Woman) \leq 1$
$\nexists\ p : Person \bullet p \in parents^+\ (\{p\})$
$\forall\ p, q : Person \bullet name(p) = name(q) \Rightarrow parents\ (\{p\}) \cap parents\ (\{q\}) = \emptyset$

HusbandsWife

FamilyDecls

$\forall\ p : Married \cap Woman \bullet$
$\quad p \in \text{dom}\ wife \land wife(p) \in \text{dom}\ husband \land husband\ (wife(p)) = p$

Op

$\Delta\ FamilyInv$

$\forall\ p: Person \cap Person' \bullet$
$\quad p \in Man \Leftrightarrow p \in Man'$
$\quad \land\ p \in Woman \Leftrightarrow p \in Woman'$
$\quad \land\ parents\ (\{p\}) = parents'\ (\{p\})$

Marry

Op

$m?, f?: PERSON$

$m? \in Man \setminus Married \land f? \in Woman \setminus Married$
$Person = Person' \land parents = parents' \land name = name'$
$wife' = wife \oplus \{m? \mapsto f?\}$

Figure 2: Z specification corresponding to Alloy model of Figure C1

Types and Sets

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

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

Temporal Constraints

Z not only has "no necessary connection with computer programming"; it does not even embody the notion of a state machine. States and state transitions are fundamentally no different, and there is no notion of constancy. It is not possible in Z to declare a state component and constrain it to be fixed. A Z specification of a file hierarchy with a fixed root, for example, would either include a template operation that does not change the root

(and which would be imported by all operations by convention), or would define *root* as a global constant, violating the modularity of the specification.

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

Navigation, Sets and Scalars

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

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

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

The Alloy constraint would only need to be changed from

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

to

```
all p, q | p.name /= q.name → no (p.parents & q.parents)
```

in which the /= operator indicates that the subformula *p.name /= q.name* is to evaluate to false when *name* does not map *p* or *q*.

Schema Roles

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

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

$$\forall FamilyInv \,.\, HusbandsWife$$

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

Relational Formulas

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

m.wife := f

would then be treated as a shorthand for two constraints:

m.wife' = w

and the frame condition

all p: Person - m | p.wife' = p.wife

Lexical and Typographic Issues

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

3 UML Version

The Unified Modeling Language (UML) [RJB99] is a combination of the notations of Rumbaugh, Booch and Jacobson. For object modelling, it provides a graphical "static structure" notation, and OCL (Object Constraint Language) [WK99], a textual notation originally developed at IBM. UML has the backing of a large consortium that includes Microsoft, Oracle, HP and IBM, and was made a standard by the Object Management Group in 1997.

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

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

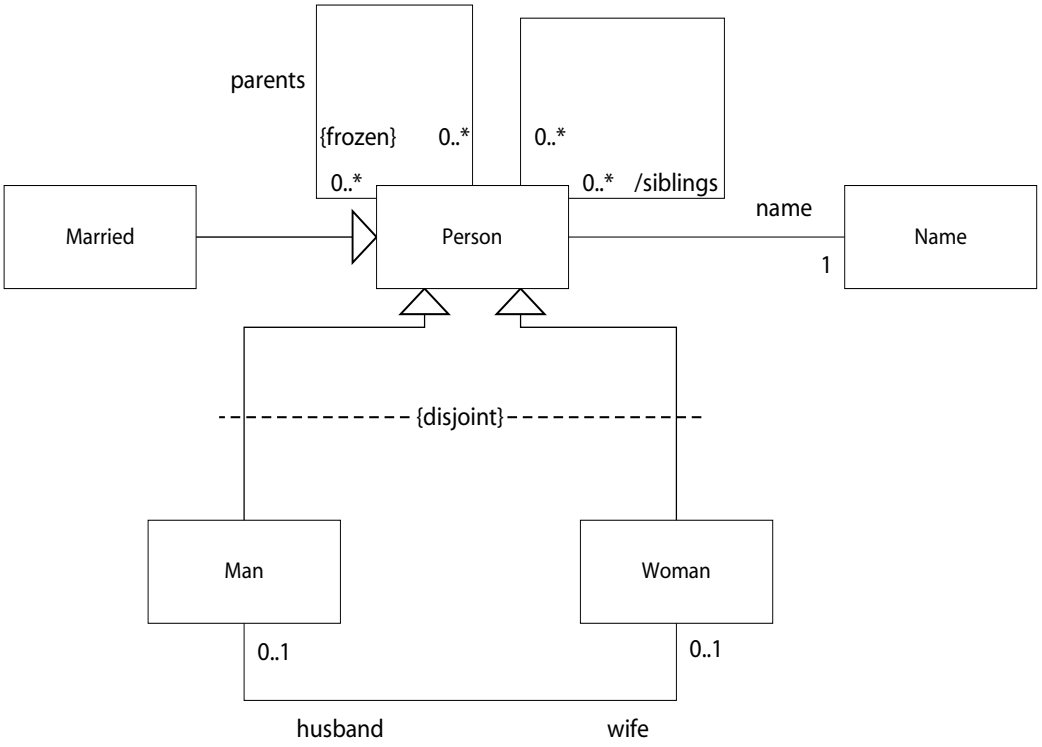

Figure 3a: Graphical part of UML model of family tree

Person

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

Man

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

Woman

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

Man :: Marry (w: Woman)

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

Figure 3b: Textual part of UML model of family tree

Gross Structure

The graphical notation of UML has no textual counterpart, so every model must include a diagram. In this case, both the diagram of Figure 3a and the textual constraints of Figure 3b are required. Moreover, those diagrammatic constraints that are expressible in the textual notation do not map to it straightforwardly. The fact that *Person* is partitioned into *Man* and *Woman*, represented in the diagram by marking *Person* as abstract and *Man* and *Woman* as disjoint, would be written

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

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

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

Contexts, Classes & Types

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

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

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

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

all p, q | p.name = q.name → no (p.parents & q.parents)

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

Classes are not treated semantically as simple sets in UML. One cannot define the men that are married as the intersection of two sets, as in Alloy or Z. Instead, a constraint about such a set must employ built-in operators to type-case and downcast objects. This can make life difficult. The following constraint from the UML semantics [Rat97, Section 9.3]

```
Collaboration

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

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

for example, would be written in Alloy as

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

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

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

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

Functions

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

Expressiveness

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

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

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

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

This does not have the desired effect, however; the operation may return the set of all messages and still satisfy its specification. Perhaps, however, it is intended to be treated as pseudocode. In this case, the desired meaning may be obtained, but the model is no longer declarative.

References

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

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

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

[Jac99] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation.* Available from: http://sdg.lcs.mit.edu/alcoa.

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

[RJB99] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

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

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