♣A.1.8 *Defining Acyclicity for an Undirected Graph*

An undirected graph can be represented as a binary relation, constrained to be symmetric. Write constraints on such a relation that rule out cycles. Here is a suitable template:

```
module exercises/undirected
sig Node {adjs: set Node}
pred acyclic () {
    adjs = ~adjs
    … your constraints here
    }
run acyclic for 4
```

A.1.9 Axiomatizing Transitive Closure

Transitive closure is not axiomatizable in first-order logic. In short, that means that if you want to express it, you need a special operator, because it can't be defined in terms of other operators. Here's a bogus attempt to do just that; your challenge is to use the Alloy Analyzer to find the flaw.

Recall that the transitive closure of a binary relation *r* is the smallest transitive relation *R* that includes *r*. Let's say *R* is a transitive cover of *r* if *R* is transitive and includes *r*. To ensure that *R* is the smallest transitive cover, we can say that removing any tuple *a*->*b* from *R* gives a relation that is *not* a transitive cover of *r*. Formalize this by completing the following template:

```
module exercises/closure

pred transCover (R, r: univ->univ) {
    … your constraints here
    }
pred transClosure (R, r: univ->univ) {
    transCover [R, r]
    … your constraint here
    }

assert Equivalence {
    all R, r: univ->univ | transClosure[R, r] iff R = ^r
    }
check Equivalence for 3
```

Now execute the command, examine the counterexample, and explain what the bug is.

(The official definition of UML 1.0 had this problem incidentally.)

In fact, for finite domains – which is how Alloy is interpreted – closure *can* be axiomatized in first-order logic. Some recent technical reports explains how to do this [16, 9]. (Thanks to Masahiro Sakai for telling me about this work.) Define a ternary relation *C* such that

```
x -> y -> z in C
```

when *y* is at some non-zero distance on a shortest path in the relation *r* from *x* to *z*. The reflexive transitive closure *R* of *r* can be expressed in terms of *C* as

```
R = {x, y: univ | x -> y -> y in C or x = y}
```

and—this is the surprising part—*C* itself can be defined by the following axioms:

```
all x, y, z, u: univ {
   x -> x -> y not in C
   x -> y -> u in C and y -> z -> u in C => x -> z -> u in C
   x -> y -> y in C and y -> z -> z in C and x != z => x -> z -> z in C
   x -> y in r and x != y => x -> y -> y in C
   x -> y -> y in C => some v: univ | x -> v in r and x -> v -> y in C
   x -> y -> z in C and y != z => y -> z -> z in C
   }
```

To check this axiomatization, just define a predicate like this:

```
pred transClosure' (R, r: univ -> univ, C: univ -> Univ -> univ) {
   ... axioms here
   }
```

and edit the assertion to read

```
assert Equivalence {
   all R, r: univ -> univ, C: univ -> univ -> univ |
      transClosure' [R, r, C] implies R = *r
   }
```

Note that the check cannot be bidirectional. Can you see why? Replace *implies* by *iff* to generate a counterexample, and explain what's going on.

A.1.10 Address Book Constraints and Expressions

In this exercise, you'll get some practice writing expressions and constraints for a simple multilevel address book. Consider a set *Addr* of ad-