Software Engineering II
Exercises

Christian Rossi

Academic Year 2023-2024

Abstract

The objective of the course is to teach the principals methods and processes of software engineering needed to develop complex and qualitative software.

The course covers the following arguments:

- Software process and its organization.
- Modelling languages.
- Requirements analysis and definition.
- Software development methods and tools.
- Approaches for verify and validate the software.

Contents

Chapter 1

Alloy examples

1.1 Address book

Imagine that we are asked to model a very simple address book. The books that contain a bunch of addresses linked to the corresponding names.

In this example we have three entities, which are: *Name*, *Addr* and *Book*. *addr* is linking *Name* to *Addr* within the context of *Book*. To indicate this relation we use the keyword *lone* that indicates that each *Name* can correspond at most one *Addr*. We can resume the previous statements as:

```
sig Name {}
sig Addr {}
sig Book {
    addr: Name -> lone Addr
}
```

This specification creates the following relations:

- Sets are unary relations.
- Scalars are singleton sets.
- The ternary relation involving the three predicates.

We can declare a new predicate with the keyword *pred*.

```
pred show {}
run show for 3 but exactly 1 Book
```

Where the second line indicates that we need to find at most three elements for every *Book*. The predicate *show* defined previously is empty and return always *true*. Now we can define a predicate with some argument, for example:

```
pred show [b:Book]{
    # b.addr > 1
}
run show for 3 but exactly 1 Book
```

The predicate (consistent) in the previous example adds a constraint on the number of *Address* relations in a given *Book*. The predicate (consistent) in the following example adds a constraint on the number of different *Address* that appears in the *Book*.

```
pred show [b:Book]{
    # b.addr > 1
    # Name.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

The predicate (inconsistent) in the following example contains the keyword *some* that indicates the existence of an element. In this case we have only one *Book* so the tool will say that no instances can be found.

```
pred add [b:Book]{
    # b.addr > 1
    some n:Name | # n.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

All the previous predicates are static because they doesn't change the signature. In Alloy there are also dynamic predicates for dynamic analysis. For example we can define a predicate that adds an *Address* and *Name* to a *Book* in the following way:

```
pred add [b,b':Book, n:Name,a:Addr]{
     b'.addr=b.addr + n -> a
}
pred showAdd [b,b':Book, n:Name,a:Addr]{
     add[b,b',n,a]
     #Name.(b'.addr) > 1
}
run showAdd
```

We can now define a predicate for the *Book* deletions.

```
pred del [b,b':Book, n:Name]{
     b'.addr=b.addr - n -> Addr
}
```

We can check if running a delete after an add returns us in the initial situation or not by using an *assertion*:

```
assert delRevertsAdd{
    all b1,b2,b3:Book,n:Name,a:Addr
    add[b1,b2,n,a] and del[b2,b3,n]
    implies b1.addr=b3.addr
}
```

While checking an assertion, Alloy searches for counterexamples. In this case we will find a counterexample so the assert will result $false$. To correct the assert we need to modify it in the following way:

```
assert delUndoesAdd{
    all b1,b2,b3:Book,n:Name,a:Addr |
    no n.(b1.addr) and add[b1,b2,n,a] and del[b1,b2,n]
    implies b1.addr=b3.addr
}
```

We can also need to get some signature. To do that we can use the Alloy functions. For example we can declare a function that search a certain *Book* and return a set of *Address*:

```
fun lookup[b:Book,n:Name]: set Addr{
    n.(n.addr)
}
```

1.2 Family relations

We now consider a family relationship tree. First of all we have to define a generic person, that can be a men or a woman.

```
abstract sig Person {
    father: lone Man
    mother: lone Woman
}
sig Man extends Person {
    wife: lone Woman
}
sig Woman extends Person {
    husband: lone Man
}
```

We have set that each $Person$ has at most one father and one mother (keyword *lone*) because we need a root for the tree. The person at the root needs to have no parents (for example they are unknown). The signature $Person$ is *abstract* because it needs to be specialized in one of the subsequent signatures, that are Man or $Woman$. Signatures by using keyword *sig* represents

a set of atoms. Before this keyword we can define the number of entities that we need (*lone*, *one* or *some*).

Definition

> The *fields* of a signature are relations whose domain is a subset of the signature. The keyword *extends* is used to declare a subset of signature.

To get the set of grandpas of a given person we can define a function like this:

```
fun grandpas[p:Person]:set Person {
    p.(mother+father).father
}
pred ownGrandpa[p:Person] {
    p in p.grandpas[p]
}
```

We have also defined a predicate that checks if the person is in the set of grandpas returned by the function *grandpas*. The problem now is that we have not set constraints on relations. To do that we need to define two new operators for binary relations:

- Transitive closure: $\hat{}r = r + r.r + r.r.r + \ldots$
- Reflex transitive closure: $*r = iden + \hat{}r$

We can now define that no one can be the father/mother of himself:

```
fact {
    no p:Person | p in p.^(mother+father)
}
```

We have also to set a constraint that if X is husband of Y, then Y is the wife of X:

```
fact {
    all m:Man,w:Woman | m.wife=w iff w.husband=m
}
fact {
    wife = ~husband
}
```

The two facts are equivalent, but the second has been written using the transpose operator. The fact can contains multiple constraints. So the previous constraints can be written in one fact. The difference between $fact$ and $pred$ is that the first are global, while the second needs to be invoked.