CS 6110 — Assignment 3

Gavin Gray u1040250

1. (30 pts) This question helps you model three theoretical notions within Alloy and study their properties. All these are discussed in Book-1.

   A (10 pts) Here are some experiments I ran to study preorders. Your task is to fill out the ellipsed portions and demonstrate what is asked.

```
-- We are defining "some old" relation 'pre'
-- and slowly endowing it with the properties that make it a preorder
--
sig S { pre  :  set S } -- (1) This defines 'pre' as a binary relation over S
fact  { some pre }      -- (2) Can you explain what this means?
fact  { state here that pre is reflexive  } -- (4) question prompt below
fact  { state here that pre is transitive } -- (5) question prompt below
assert preAndPreinvIden
{ FALSELY Assert that the intersection of pre and its inverse is identity. }
-- (6) question below
check preAndPreinvIden for exactly 3 S
-- (7) Report on the result of this check
run {} for exactly 4 S
-- (8) If the check in (7) fails, comment (7) and run this to diagnose why
```

   (1) `pre` is introduced as a "plain old" relation
   (2) Explain what this assertion does for `pre`
   (3) At this juncture, "run" for "`exactly 4 S`" and show the models generated.
   (4) How did you endow 'pre' with the property that it is reflexive? Explain.
   (5) Explain how you endowed 'pre' to be transitive
   (6) How did you falsely assert that the intersection of pre and its inverse is identity?
   (7) Did this check pass? You can use a pull-down menu and run this check. If the check in (7) failed, look at the counterexample and explain why it failed.
   (8) Also run this "run" statement and see the instance generated. Does the instance generated provide another explanation as to why the above `check` failed?

   Student Response:

   (1)

   (2) Including the keyword `some` before the set means that it cannot be empty.

   (3) At this point any set of nodes is valid as long as each one has a nonempty

set of relationships to the others (i.e. there must be at least one line coming out of it).

(4) To say that S is reflexive I used the following fact

```
fact  { all s: S | s->s in pre  }
```

which states that `s -> s`, the relationship from s to itself, must be in the set of relationships 'pre'.

(5) To say that S is transitive I used the following fact:

```
fact  {
    all x,y,z: S | ((x -> y in pre) and (y -> z in pre))
                        implies (x -> z in pre)
}
```

which states simply that iff 'x' is related to 'y', and 'y' is related to 'z', then 'x' must be related to 'z'.

(6) The assertion that the intersectino of pre and its inverse is identity is as follows:

```
assert preAndPreinvIden
    { all s: S |  (s.pre & s.~pre) = s.iden }
```

(7) The counterexample given for the above assertion is, for all three `s: S`, `s.pre = S`. We can see that the intersection of the relational inverse with itself, is just the original relation, not the identity.

(8) Running without the assertion and playing with the evaluator panel, we can see that the following statement holds: `(S.pre & S.~pre) = (S<:univ)` and running this as an assertion also passes.

B (10 pts)

- Repeat the above steps, except also show you modeled `anti-symmetry`.
- Now show that the assertion above is true.
- Instead of "check," show some "run" results to convince me (and yourself) that the instances are indeed indicative of the assertion check succeeding.

Student Response:

(1) anti-symmetry can be modeled as follows:

```
fact{ all x,y: S | (x != y and x->y in pre) => y->x not in pre }
```

(2) Running with the same check as above shows no counter examples generated. Some generated instances are included below:

Figure 1: Generated instances

C (10 pts) The Schröder-Bernstein Theorem (SBT) says that for two sets A and B, if there is a function from A to B that is 1-1 and also a total function (defined everywhere in A) and if there is a function from B to A that is 1-1 and also a total function (defined everywhere in B) then A and B have the same cardinality. Show this in Alloy as follows:

```
sig A { f : lone B } -- f is a function                              -- (1)
fact { express that f is 1-1 here }                                  -- (2)
fact { express that f is "total" i.e. defined for all of A here }    -- (3)
sig B { g : lone A } -- g is a function                              -- (4)
fact { express that g is 1-1 here }                                  -- (5)
fact { express that g is "total" i.e. defined for all of B here }    -- (6)
assert SBT { #A = #B }                                               -- (7)
check SBT for 10                                                     -- (8)
```

Basically, encode all that is needed to demo that the above assert passes.

(1) Why does this ensure that `f` is a function?
(2) How did you express this?
(3) How did you express this?
(4) Why does this ensure that `g` is a function?
(5) How did you express this?
(6) How did you express this?
(7) What does this assertion say?
(8) Did this check pass? Run the model a few times and look at the instances generated, and corroborate your answer.

Student Response:

(1) A function is a mapping from values to a value. By restricting the associated set 'f' to be at most one value (with the keyword `lone`) we are achieving this

functionality.

(2) To express that 'f' is 1-1 I gave the fact:

```
fact { no p, q: A | p != q and some p.f & q.f }
```

which says that no two mapping from $A \to B$ can be the same for separate instances of A.

(3) To express that 'f' is *total* I gave the fact:

```
fact { all a: A | some a.f }
```

which states that for all instances of 'A', 'f' cannot be empty. This could have also been achieved by including in the signature of 'A' that 'f' is `one`.

(4) 'g' is assured to be a function for the same reason that 'f' is a function. It is a mapping from 'B' instances to 'A' instances.

(5) I expressed this the same as the condition on 'A', namely:

```
fact { no p, q: B | p != q and some p.g & q.g }
```

(6) Instead of expression this as I did for 'A', I changed the signature of 'B' to `sig B { g : one A }`.

(7) The assertion states that the cardinality of 'A' and 'B' must be the same.

(8) Yes, the check passes. By generating a few instances we can see that there is a clear 1-1 mapping between the sets 'A' and 'B'.

2. (10 pts) Read and summarize Pamela Zave's paper on comparing Alloy and SPIN in a page. The paper is `spin-versus-alloy-comparison*` in the "Files" area on Canvas. A glance at the paper `TSE_Chord*` will also help.

   Student Response: In her paper **A Practical Comparison of Alloy and Spin**, author Pamela Zave gives a comparison of the popular model checking tools, Alloy and Spin. The driving force behind this research is to encourage more academic and industry researchers to involve model checking in their process. Formal methods and model checking is largely not used due to the following barriers: time to learn the tools, ambiguity as to which tool to use. The demonstration of using these lightweight yet powerful tools should help show readers that it is possible to do model checking without spending years mastering the intricacies of these methods. To compare and contrast these tools Zave seeks to check the popular Chord network structure. Neither Alloy nor Spin provide a perfect user experience when modeling a Chord network. The Results drawn by Zave are as follows:

   The advantages of Spin over Alloy were: not necessary to know a sufficiently strong global invariant, and supports progress assertions. The advantages of Alloy over Spin were: half the startup time, safety assertions are declarative rather than procedural, and can be used for a convincing proof of correctness. At the beginning of the case study, Zave held a hypothesis for what was expected to be advantageous, however, some of the properties proved to not be helpful at all! For Spin it is typically regarded as the better choice for network protocols, and has automated verification of progress assertions. For Alloy the automated visualization of examples as graphs proved to be unhelpful and the graph was re translated by hand (similar as needed by the error trace given by Spin).

   Zave reluctantly concludes by stating a lightweight recommendation for modeling network protocols as a "best practice" was not reached through this study. Regardless, the paper provided a good contrast to how a complex network protocol can be modeled in both Alloy and Spin, despite the two tools having a vastly different view of model checking.

3. (20 pts) Run through the entirety of Chapter-2, showing the runs for every partial run of the address book. *Provide a transcript of the run similar to the session I demonstrated today.* Provide a highlight of what you observed along the way. You don't have to dump all of what your runs produced. Just convince me that you did incrementally run through Chapter-2 and indeed observed the highlights of the printout. The PDF for Chapter 2 (scanned) will be kept on Canvas in the `ScannedChapter2Alloy/` folder by midnight of Feb 8, 2021.

Student Response: You can find the full code (for this first model) in the file `addressbook1.als` included in the assignment zip.

To begin, we will start with a few definitions:

```
sig Name, Addr {}
sig Book { addr: Name -> lone Addr }
pred show (b: Book) { }
```

The above is defining our signals *Name, Addr*, and *Book* where a book has a set of mappings from Name to at most one Addr. Running the above with `run show for 3 but 1 Book` we get an instance such as in 2. I've included the table view because I was (am) a little

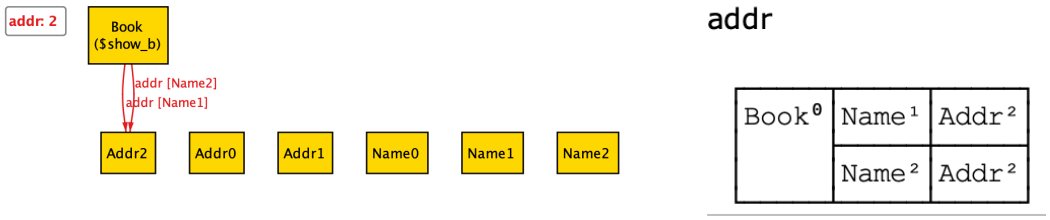


Figure 2: Generated instances

confused why the graph view doesn't show the relationship between Names and Addrs. We can now add some constraints to the predicate `show` to indicate that we want at least 1 name address mapping in the book and there can be multiple names.

```
pred show (b: Book) {
    #b.addr > 1
    -- predicate inconsistent!
    -- some n: Name | #n.(b.addr) > 1
    #Name.(b.addr) > 1
}
```

We can now add operations to the book, such as adding a name. This can again be done through a predicate:

```
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
}
```

The `add` predicate is saying that given two books b, b', where b' is equivalent to b with the addition of the new mapping from `n -> a`. A generated instance of this is shown below in 3. You can notice that `Book0` is b' as it has the addition of the `Name1 -> Addr1`

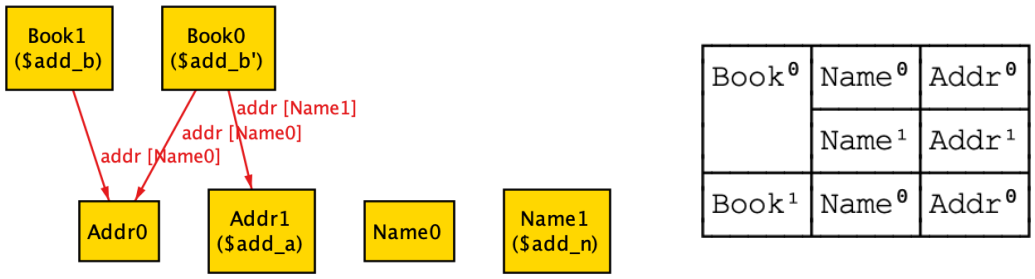


Figure 3: Generated instances

mapping. We can give the model even more functionality and even try and generate counterexamples with the following additional code.

```
pred del(b, b': Book, n: Name) {
    b'.addr = b.addr - n->Addr
}
fun lookup(b: Book, n: Name): set Addr {
    n.(b.addr)
}
assert delUndoesAdd {
    all b,b',b'': Book, n: Name, a: Addr |
        no n.(b.addr) and
        add[b,b',n,a] and
        del[b',b'',n] implies
            b.addr = b''.addr
}
check delUndoesAdd for 3
assert addIdempotent {
    all b,b',b'': Book, n: Name, a: Addr |
        add[b,b',n,a] and add[b',b'',n,a] implies b'.addr = b''.addr
}
```

```
check addIdempotent for 3
assert addLocal {
    all b,b': Book, n,n': Name, a: Addr |
        add[b,b',n,a] and n != n' implies lookup[b, n'] = lookup[b',n']
}
check addLocal for 3
```

To improve the model and add more functionality (similar to an actual address book), we will start anew. The full code for this can be found in `addressbook2.als`. This new model will reuse some features and ideas from the previous, but let's start with some definitions (as always).

```
abstract sig Target {}
abstract sig Name extends Target {}
sig Addr extends Target {}
sig Alias, Group extends Name {}
sig Book { addr: Name -> Target }
    { no n: Name | n in n.^addr }
```

Similar to the last model a `Book` is just a mapping, this time from Names to Targets where a Target is either an Addrs, or a Name (which can itself be a Group or Alias). This extra layer of abstraction helps us achieve more functionality that a real address book might have. To run what we currently have we can make a simple predicate like before and run it which generates something as in 4.

```
pred show(b: Book) { some b.addr }
run show for 3 but 1 Book
```

We can now add a few more predicates to ensure that a name is mapped to an address eventually, in addition to the ones that we had for the first model. This brings us to the full code for the second address book model (again, which can be found in `addressbook2.als`. An example instance from this code is seen in 5.

For the last model, the full code can be found in `addressbook3.als`. This code starts from the same base that `addressbook2.als` was left in but adds the necessary mechanisms to look into empty lookup scenarios. To begin we include the `util/ordering` module and instantiate it with an ordering of Book (`[Book]`). We can then add traces by

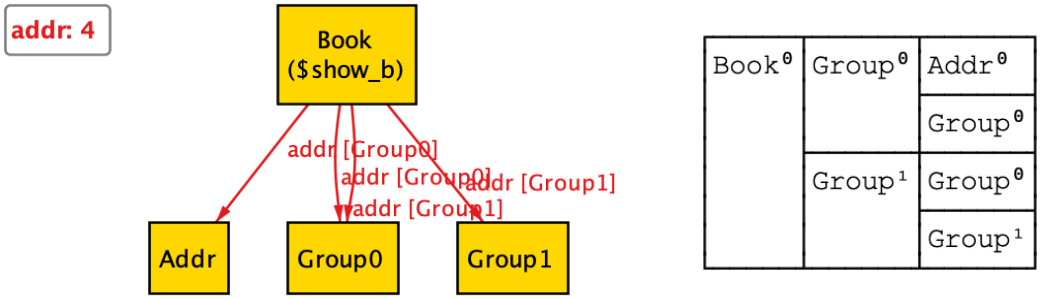


Figure 4: Generated instances

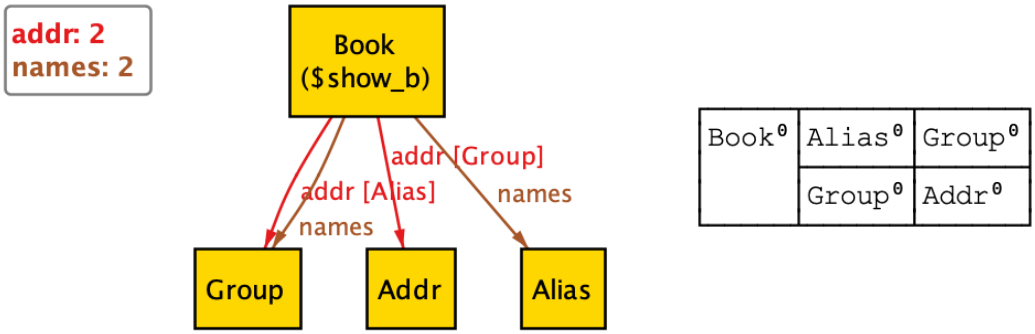
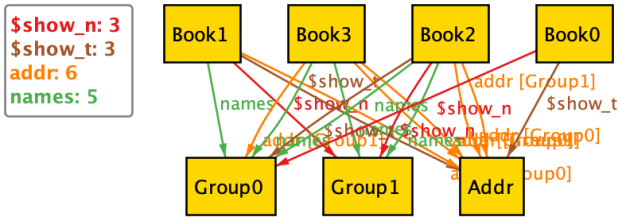


Figure 5: Generated instances

starting with an initial empty book and each subsequent book is related to the preious through an addition or deletion. The code alterations added the following:

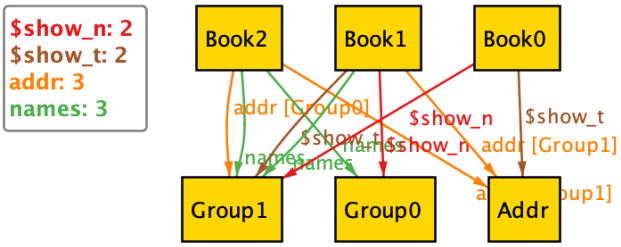
```
pred init (b: Book) { no b.addr }
fact traces {
    init[first]
    all b: Book - last | let b' = next[b] |
        some n: Name, t: Target | add[b,b',n,t] or del[b,b',n,t]
}
```

The first generated instance is seen in 7. We can note that running the assertion `lookupYields` generates a counterexample because we are allowed to add an Alias to a Group even if it doesn't refer to an Addr! To fix this, we can add this constrain to the `add` and `del` predicates. Now running the assertion yields no counterexample, we can see a final generated instance in **??**.



Book¹	Group⁰	Addr⁰
Book²	Group⁰	Addr⁰
	Group¹	Addr⁰
Book³	Group⁰	Addr⁰
	Group¹	Addr⁰
		Group⁰

Figure 6: Generated instances



Book¹	Group¹	Addr⁰
Book²	Group⁰	Group¹
	Group¹	Addr⁰

Figure 7: Generated instances

4. (20 pts) Read and summarize Baugh et al's paper on modeling sparse matrices in a page. The paper is `dyer-correctness*`

   Student Response: In the paper *Bounded Verification of Sparse Matrix Computation* by Baugh et al, we get a view on how to model and reason about the behavior of sparse matrices. Importantly, a state-based approach is developed to help relate more directly with FORTRAN and C++ codes.

   I don't know how much detail I can go into when summarizing the techniques used (as many of it went over my head), however, the approach was largely incremental. One question that I had when reading the paper is related to how Baugh et al alludes to "common" idioms that Alloy users default to when trying to represent some functionality. Is there a reason that these idioms couldn't be incorporated into the Alloy language? Would it break/violate some of the set theory that Alloy relies on?

   To conclude, the group positively reflects on the methods developed as they truly shared basic elements with state based code approaches. Alloy has trended to be a favorite in the literature that we've read thus far and has the ability to model a wide variety of systems.

5. (20 pts) Read and summarize Baugh et al's paper on modeling coastlines in a page. The paper is `baugh-scp*`.

Student Response: In this paper Professor Baugh explores the verification of hurricane storm surge detection software through the use of Alloy. He first models finite meshes using triangles (a common scientific computing technique) which van be represented in Alloy simply through the following:

```
sig Mesh { triangles: some Triangle, adj: Triangle -> Triangle }
sig Triangle { edges: Vertex -> Vertex }
sig Vertex { }
```

The above comprises the signal definition of the Mesh and naturally there are a few facts and predicates used to ensure it's validity (such as forming a ring, unique triangles, etc . . .). I thought how the mesh was defined to satisfy Euler's formula was very interesting, namely:

```
fun undirectedEdges [m: Mesh]: Int {
    let e = m.triangles.edges | minus[#e, div[#(e & e), 2]]
    }
fact{ allm: Mesh |
    let T = #m.triangles, E =undirectedEdges[m], V =#dom[m.triangles.edges]
        | minus[T, 1] = minus[E, V] }
```

I think it is important to note how Alloy can model state and time transitioned models. I noticed in both the address book example in chapter 2 of the Alloy book as well as here in Baugh's paper that an "ordering" is used. I think this idea is very intuitive and easier to think about than Spin's concurrent model (which I do find a little confusing at times). Lastly, Baugh demonstrates how easy it is to look for specific instances of a model using predicates. This is done when he wants to find a dry element with three wet nodes. The advantage to Alloy here is a model satisfying this predicate can be generated in a fraction of the time as compared to using ADCIRC.