CS 6110, Spring 2022, Assignment 5

Given 2/22/22 – Due 3/3/22 by 11:59 pm via your Github

NAME: **UNID:**

1. (10 points) **Read and summarize Jackson's article on Alloy**

Alloy tool can be used to explore models that arise in different software design. We can use simple logic relation in alloy to build structures that are dependent on relations. We can simulate designs and find flaws in our structure using alloy as it uses SAT technology. It is complementary to model checking tools.

- The software we design should have the properties we expect. Informal representation give inconsistent interpretations and they cannot be analysed easily. By using formality, we can minimize the cost of ambiguity and can get quicker feedbacks.
- Theorem provers are mechanical aids for constructing mathematical proofs. We need expressive logic and sound proofs which are hard to automate. Hence a lot of effort and expertise id required for this process.
- Model checker provides push-button automation, and the user have to just provide the design and property that needs to be checked. They provide counter examples when properties are not true. They also allow dynamic properties to be expressed. But model checkers have a problem of state explosion.
- Alloy is built to provide the advantages and eliminate the problems of model checker. Model checkers are used for parallelism and simple states, but do not provide support for rich structures like trees, lists, tables and graphs.

Alloy innovations

- Relational logic are used in alloy for both describing designs and properties.
- Alloy incorporated the power of Z language in which the sets and relations are used for software design.
- Alloy allows only first order structures, thus ruling out sets over sets and relations over sets.
- The key operator us relational join, which works with relations of any arity. With alloy we can describe the universe of objects as a classification of tree. Alloy runs all small tests. The designer can specify the scope that bounds for each of the specifications given. Alloy translates the design problem to a satisfiability problem whose variables are not relations but simple bits.

How to use alloy analyser and an example

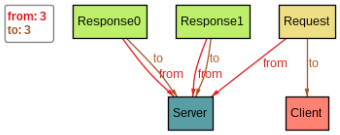
- A structure is represented as a signature "sig" which can be a collection of points and we can also specify the property that these points can hold. For example in the screenshot we see two signature EndPoint and HTTPEvent. The HTTPEvent consists of collection of points to and from. The run command will execute to produce an instance of the httpevent that has been created.

```
sig EndPoint {}
sig HTTPEvent {
    from, to : EndPoint
}
Execute
run{}
```

- We can further classify the EndPoints as clients and servers. The HTTPEvents are also classified as requests and response to obtain a graph that shows the connection between clients, servers, response and request.

```
abstract sig EndPoint {}
sig Client, Server extends EndPoint {}
abstract sig HTTPEvent {
    from, to : EndPoint
}
sig Request extends HTTPEvent {}
sig Response extends HTTPEvent {}
Execute
run{}
```

- We can improve the theme in alloy by giving different colors to each object and the relations. The figure below shows the martha palette in the theme and giving different colors by changing the general graph settings.



- We see in the above figure that the request is to the server and response is to and from server. This can be constrained by using a fact as shown the figure below.

```
abstract sig EndPoint {}
sig Client, Server extends EndPoint {}
abstract sig HTTPEvent {
    from, to : EndPoint
}
sig Request extends HTTPEvent {}
sig Response extends HTTPEvent {}
fact {
    Response.to + Request.from in Client
    Response.from + Request.to in Server
}
Execute
run{}
```

- To create a relation between request and response, we set a property in the reponse to point out how request is related to response. To obtain exactly one response for each request we set a fact that each response is for exactly one request. We can also add a constraint of each property that we want to see in the run command.

```
abstract sig EndPoint {}
sig Client, Server extends EndPoint {}

abstract sig HTTPEvent {
    from, to : EndPoint
}
sig Request extends HTTPEvent {
    response:Response
}
sig Response extends HTTPEvent {}

fact {
    Request.from + Response.to in Client
    Response.from + Request.to in Server
    all r:Response | one response.r
    all r:Response | r.to = response.r.from
}

run {
   some response
}
```

- To see if any property is true for our model, we use check command. For example we can check that any request if to goes as the reponse to the same request it originally comes from. If the alloy finds a counter example then the claimed property does not exactly hold. This means we need to constraint out facts again. Once the facts are all fixed then the check will not produce any counter example.

```
check {
    all r: Request | r.to = r.response.from
}
```

- If alloy produce no counter example then produce the core for the model. The core produce the instance that are used to check the property that has been provided.

2. (25 points) **Take the tutorial on `https://www.doc.ic.ac.uk/project/examples/2007/271j/suprema_on_alloy/Web/index.php`**

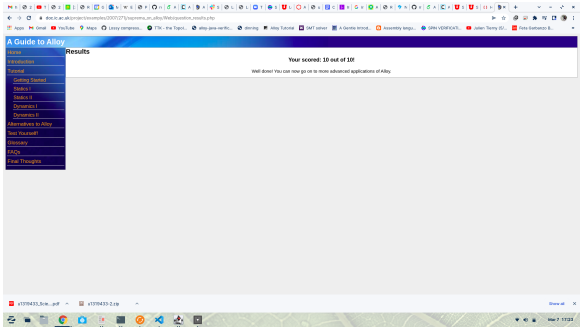
- **Statics I** The tutorial is to use alloy to model the implementation of a Queue. Every queue has a root node and a reference to next node in the queue. We decalre two signatures in the alloy one for the queue and other for the node. The queue consists of the root which is a lone Node and the node consists of next which is also a type of lone Node. Lone means zero or one. To produce any instance of the Queue we can use predicate with no constraints. To get samples of predicate we use run command. We can force the run command to produce instance of a particular size. We can add facts to constrain our model.

- **Statics II** The $*$ operator represents the reflexive transitive closure. The operator $\hat{}$ (hat) is use to represent non-reflexive transitive closure. Ally provides us the facility to visualize different instances of the model to find errors. This helps us in substituting specific constraint to obtain the correct model.

- **Dynamics I** Abstract ensures that there cannot be direct instances of the object. In the predicate we can constraint what type of instances to produce. We use # to produce number of any type of signature. Exactly is to constraint a particular number of instance to produce.We can use assert to restrict the model to have a particular assertion. We can then check the assertion to see if the model produces an output or not. If yes then the model is not correct otherwise the model follows the assertion. All is used for the all the elements in the set and some for some elements in the set.

- **Dynamics II** + is a set union operator. This means that the set of values after is equal to the union of the set of values before with extra mapping from k to v. fun is used to create a function in alloy and arguments to a function are enclosed in square brackets. Dot (.) operator is used to return all values that are associated with a particular object.

- **Evidence of the test**



3. (40 points) First, deep-read the Alloy "Red book" linked on the class website from the beginning till Page 80. Summarize the concepts you find on these pages in four pages. Thus, there are three tree models below. Study them and do the problems stated below them:

   Q1 : no node in the tree that points to the root
   Q2 : no node in the tree that points to another node such that it makes a loop (non-transitive)
   We cannot use $*$ because this means reflexive transitive closure, whereas $\hat{}$ means transitive closure. If we use $*$, it will make nodes point to itself. No instances are found if we use $*$ instead of $\hat{}$.

   Q3 : all the nodes except the root has a root, there is only one root and the tree is one connected tree.
   We need Node - root so that the relation applied is not applied to the root. Hence there is no root for the root node.

   Q4 : no node points to itself (non-reflexive).
   If we leave disj then we end up getting just one root node. This is because all the properties are too strict and hence we just get a tree consisting of only one node.

   Q5 :no transitive relation between nodes

   (a) Issue `check {GGTree iff DJTree} for 5` and see if the definitions are equivalent.
   Since no instances are found, hence the two definitions, GGTree and DJTree are equivalent

   (b) Just break the Q-1 to have $*$ and not the correct ^. Do the "`iff`" check above. Do you get an understandable counterexample? Describe it.
   After breaking the $*$ in Q1, we end up getting counter examples. This means that there is a reflexive relation between the nodes and they can point to the root. Such a tree is not possible and hence we end up getting counter examples that does not exists in GGTree but exists in DJTree.

   (c) Issue `check {CostelloTree iff DJTree} for 5` and see if the definitions are equivalent.
   Since no instances are found, hence the two definitions, CostelloTree and DJTree are equivalent
   The codes are available on Github as tree.als and broken_tree.als

- Some issues might look trivial and are not easy to identify. Alloy provides you the facility to get a graphical representation of the system consisting of nodes and edges, defined over different functions to show the relationship.
- In alloy model, signatures are used which consists of objects. We can map any of the objects by using a field name and then defining a relationship that exists on it. Keywords like lone can be used to show multiplicity.
- Predicate and run commands are used to find the instance of the possible states. We can use use the pound to show the instance of a particular link constraint to a specific number.
- Different dynamic operators like add can be used to produce a specific constraint. Even though Alloy is declarative, it can still be executed like operational language. This is done using the run command.
- Other dynamic operations are minus (-), dot (.). - is used for deletion and . can be used to find a particular object in a signature.
- Assert is used to insert an assertion. To check the assertion we use check command. If we get a counter example then the assertion is not valid.
- We can create alias for an object in Alloy. This is done using extends keyword. The hat operator is used for transitive closure. We can write facts that are true for signatures.
- We can create functions using keyword fun. Overall, alloy do not use any unfamiliar or complicated mathematics and it is easy to these symbols as these are based on simple basic logic and a special dot operator for navigate along relations, which is similar to deferencing in java.
- We can write partial description of any operation that allows different behaviours.
- With assertions we can test everything and we do not need any extra test case to check the results.
- Navigation expressions are commonly used as multiplicity constraints are very common. The predicate calculus is commonly used in comprehension expressions. These are also used when writing concrete and straightforward constraints.
- Relational calculus are used for some commonly recurring constraints that can be expressed more concisely, example no $\hat{r}\&$ iden to say that the relation r is acyclic.
- An atom is a primitive entity that is indivisible, immutable, and uninterpreted.

- A relation is a structure that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms. A relation can have any number of rows, called its size. Any size is possible, including zero. The number of columns in a relation is called its arity, and must be one or more. Relations with arity one, two, and three are said to be unary, binary, and ternary. A relation with arity of three or more is a multirelation.

- A unary relation with only one tuple, corresponding to a table with a single entry, represents a scalar. A relation with no tuples is empty. A unary relation with at most one tuple—that is, a relation that is either a scalar or empty—is called an option.

- Composite objects can be created with atoms for the components and a relation to bind them together. We can model mutation, in which the value of an object changes over time, by separating the identity of the object and its value into separate atoms, and relating identities, values and times.

- Atom names never appear in models; they're only used to describe instances produced by simulation or checking. All structures are relations, and a set is simply a relation all of whose tuples contain only one element. The restriction to flat relations makes the logic more tractable for analysis.

- There is a loss of expressive power in the restriction to flat relations, which can be work around. Relations have finite size and arity. Multirelations are used because relations are flat rather than nested, arities greater than two are very common. To model execution traces of a system whose state involves relationships will require ternary relations: two columns for the relationship at a given time, and an additional column for the time.

- An unordered relationship can be represented in different ways. The simplest way is to use a relation r (ordered, as always), and add a constraint r = $\tilde{r}$ that makes it symmetric—the same forward and backward.

- A binary relation that maps each atom to at most one other atom is said to be functional, and is called a function. A binary relation that maps at most one atom to each atom is injective.

- The domain of a relation is the set of atoms in its first column; the range is the set in the last column.

- Particular values of sets and binary relations can be shown graphically in a snapshot. Alloy creates a node for each atom, and draw an arc for each tuple connecting the nodes corresponding to the first and second atoms in the tuple. To show several relations, you label each tuple arc with the relation it belongs to. Sets can be shown in two ways: either by an extra label in a node naming a set it belongs to, or by drawing a labelled contour around some nodes.

- Multirelations can be shown as graphs by projecting out one or more columns. Projection takes two steps. Suppose just one column is being projected out. In the first step, the column is moved to the front, so that it becomes the first column of the relation; each tuple is permuted accordingly. In the second step, the relation is split into an indexed collection of relations. For each atom that appears in the first column, we associate the relation consisting of all those tuples that begin with that atom, but with the atom removed.

- There are three constants: none, univ and iden. None and univ , represents the set containing no atom and every atom respectively, are unary. To denote the empty binary relation, we can write none $\rightarrow$ none , and for the universal relation that maps every atom to every atom, univ $\rightarrow$ univ. The identity relation is binary, and contains a tuple relating every atom to itself.

- The set operators are $+$ (union), $\&$ (intersection), $-$ (difference), in (subset), $=$ (equality). These operators can be applied to any pair of relations so long as they have the same arity. Because scalars are just singleton sets, the braces used to form sets from scalars in traditional mathematical notation aren't needed.

- The relational operators are : $\rightarrow$ (arrow - product), . (dot- join), [] (box-join), $\tilde{}$ (transpose), $\hat{}$ (transitive closure), * (reflexive-transitive closure) $<:$ (domain restriction), $:>$ (range restriction), $++$ (override).

- The arrow product (or just product) p $\rightarrow$ q of two relations p and q is the relation you get by taking every combination of a tuple from p and a tuple from q and concatenating them.

- The quintessential relational operator is composition, or join. The dot join (or just join) p.q of relations p and q is the relation you get by taking every combination of a tuple in p and a tuple in q , and including their join, if it exists.

- The transpose $\tilde{.}$r of a binary relation r takes its mirror image, forming a new relation by reversing the order of atoms in each tuple.

- A binary relation is transitive if, whenever it contains the tuples a b and b c , it also contains a c , or more succinctly as a relational constraint: r.r in r. The transitive closure $\hat{.}$r of a binary relation r , or just the closure for short, is the smallest relation that contains r and is transitive.

- The restriction operators are used to filter relations to a given domain or range. The expression s $<:$ r , formed from a set s and a relation r , contains those tuples of r that start with an element in s . Similarly, r $:>$ s contains the tuples of r that end with an element in s . Restrictions can be applied to relations of any arity of two or more, but are most often applied to binary relations.

- The override p ++ q of relation p by relation q is like the union, except that the tuples of q can replace the tuples of p rather than just augmenting them. Any tuple in p that matches a tuple in q by starting with the same element is dropped. The relations p and q can have any matching arity of two or more.
- Operators have a standard precedence ranking so that constraints aren't marred by masses of parentheses. The ranking follows the usual conventions: unary operators (closure, transpose) precede binary operators; product operators (such as dot and arrow) precede sum operators (plus, minus, intersect).
- We've seen how to make a constraint from two expressions using the comparison operators in and = . Larger constraints are made from smaller constraints by combining them with the standard logical operators and by quantifying constraints that contain free variables over bindings.
- There are two forms of each logical operator: a shorthand and a verbose form: not (negation), and (conjunction), or (disjunction), implies (implication), else (alternative), iff (bi-implication).
- A quantified constraint takes the form Q x: e | F where F is a constraint that contains the variable x , e is an expression bounding x , and Q is a quantifier. Quantifiers in alloy: all, some, no, lone, one.
- When an expression appears repeatedly, or is a subexpression of a larger, complicated expression, you can factor it out. The form let x = e | A is short for A with each occurrence of the variable x replaced by the expression e . The body of the let, A , and thus the form as a whole, can be a constraint or an expression.
- Comprehensions make relations from properties. A declaration introduces a relation name. A constraint of the form relation-name : expression is called a declaration. Multiplicities are expressed with the multiplicity keywords: set, one, lone, some.
- Suppose the declaration looks like this: r: A m $\rightarrow$ n B where m and n are multiplicity keywords (and where A and B are, for now, sets). Then the relation r is constrained to map each member of A to n members of B , and to map m members of A to each member of B.
- Declarations usually introduce new names, but they can also be used to impose constraints on relations that have already been declared, or on arbitrary expressions.
- Multiplicities can be nested. Suppose you have a declaration of the form r: A $\rightarrow$ (B m $\rightarrow$ n C) This means that, for each tuple in A , the corresponding tuples in B $\rightarrow$ C form a relation with the given multiplicity.
- The operator pound applied to a relation gives the number of tuples it contains, as an integer value. The following operators can be used to combine and compare integers: +, -, =, <, >, =<, >=.

4. (25 points) Write your own quicksort and broken quicksort in C with klee assertions

The code is available on Github quicksort.c and broken_quicksort.c. Here is the output screenshot for quicksort and its broken version (as decribed in the question)

```
klee@e26a02c37faa:~/klee_src/examples/sort$ klee -posix-runtime quicksort.bc -sym-stdin 10
KLEE: NOTE: Using POSIX model: /tmp/klee_build60stp_z3/Debug+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/klee_src/examples/sort/klee-out-6"
KLEE: Using STP solver backend
warning: Linking two modules of different target triples: quicksort.bc' is 'x86_64-unknown-linux-gnu' whereas 'klee_init_env.bc' is 'x86_64-pc-linux-gnu'

KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling external: syscall(4, 94591355280032, 94591354014240) at /tmp/klee_src/runtime/POSIX/fd.c:528 12
KLEE: WARNING ONCE: calling __klee_posix_wrapped_main with extra arguments.

KLEE: done: total instructions = 40540
KLEE: done: completed paths = 120
KLEE: done: generated tests = 120
```

Figure 1: Klee output for quicksort.c

```
klee@e26a02c37faa:~/klee_src/examples/sort$ klee -posix-runtime broken_quicksort.bc -sym-stdin 10
KLEE: NOTE: Using POSIX model: /tmp/klee_build60stp_z3/Debug+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/klee_src/examples/sort/klee-out-5"
KLEE: Using STP solver backend
warning: Linking two modules of different target triples: broken_quicksort.bc' is 'x86_64-unknown-linux-gnu' whereas 'klee_init_env.bc' is 'x86_64-pc-linux-gnu'

KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling external: syscall(4, 94752764287704, 94752763071008) at /tmp/klee_src/runtime/POSIX/fd.c:528 12
KLEE: WARNING ONCE: calling __klee_posix_wrapped_main with extra arguments.
KLEE: ERROR: broken_quicksort.c:54: ASSERTION FAIL: temp1[i] <= temp1[i+1]
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING: Maximum stack size reached.

KLEE: done: total instructions = 622275
KLEE: done: completed paths = 41
KLEE: done: generated tests = 10
klee@e26a02c37faa:~/klee_src/examples/sort$
```

Figure 2: Klee output for broken_quicksort.c