Homework #1: Formal Specifications

CS 706, Fall 2012

assigned Wednesday, September 19, 2012
due Wednesday, **October 3, 2012**
maximum possible score: **100 points**

General Instructions

Homework assignments are for individuals. Work alone. You may discuss broad concepts and ideas with other students, or review the general form of an algorithm. Stay away from the concrete details of any specific homework problem, though. Class projects are for groups, but homework assignments are for individuals.

Deadlines

Your solutions are due by 11:59pm on Wednesday, October 3. You may submit your solutions up to two days late with a cumulative 10% score penalty per day. Solutions will not be accepted after 11:59pm on Friday, October 5.

Submitting Your Solutions

Submit your work via the course Moodle site. You should upload the following files:

- `solutions.pdf`, a PDF document with your solutions to the problems in sections 1 to 2.
- `binaryTree.als`, your Alloy solution to problem 3.1.
- `searchTree.als`, your Alloy solution to problem 3.2.
- `avlTree.als`, your Alloy solution to problem 3.3.

You may submit other files if they are a necessary part of your solution. For example, if you have an important illustration which you could not incorporate into `solutions.pdf`, or one of your Alloy modules is split across several files, you can include these separately. In the case of illustrations, we prefer that you incorporate them all directly into `solutions.pdf` whenever possible, but this is not a class about typesetting. If you do have additional files, be sure to clearly tell us to examine them where appropriate. For example, write "see the diagram in file `foo.png`" at the appropriate place in `solutions.pdf`. It is your responsibility to help us find all of the pieces of your solution. If your instructions are unclear, or if we are unable to view supporting files with moderate effort, you may lose points.

1 Computation Tree Logic

Let model M be defined by the following Kripke structure with atomic propositions a and b. Each box gives the name of a state (s_i) and the subset of atomic propositions which are true in that state.

$$\begin{tikzcd}
{} \arrow[r] & s_0\colon \{a\} \arrow[d] & s_1\colon \{a\} \arrow[r] & s_2\colon \{b\} \arrow[d] \\
& s_3\colon \varnothing \arrow[r] & s_4\colon \{a\} \arrow[ul] \arrow[u] \arrow[r] & s_5\colon \{b\} \arrow[ul] \arrow[loop right]
\end{tikzcd}$$

The next three problems ask you to identify the states of M that satisfy some CTL formula, and decide whether M itself satisfies the same formula. Use the systematic CTL checking algorithm described in class; solving using your intuition alone will earn no points. Show your work, not just a final answer. For example, if you rewrite a formula into an equivalent form, show the rewritten form. List the states satisfying each subformula as you build up to an answer for the original formula. Full points require both a correct final answer as well as work clearly demonstrating the systematic, algorithmic steps leading to that answer.

Problem 1.1 (5 points):
Identify the set of states $\{s : M, s \vDash \mathbf{E}(a\,\mathbf{U}\,b)\}$. Does $M \vDash \mathbf{E}(a\,\mathbf{U}\,b)$?

Problem 1.2 (10 points):
Identify the set of states $\{s : M, s \vDash \mathbf{AG}\,\mathbf{E}(b\,\mathbf{U}\,a)\}$. Does $M \vDash \mathbf{AG}\,\mathbf{E}(b\,\mathbf{U}\,a)$?

Problem 1.3 (10 points):
Identify the set of states $\{s : M, s \vDash \mathbf{AF}\,a\}$. Does $M \vDash \mathbf{AF}\,a$?

2 Linear Temporal Logic

Consider a Kripke structure M for which $\{p\}$ is the entire set of atomic propositions. Suppose I follow a path in M, but instead of giving you the name of each state, I only reveal whether p or $\neg p$ is true in each state along the path. Each path through M yields an infinitely long string on the alphabet $\Sigma \equiv \{p, \neg p\}$. For example, one path might begin $\langle p, p, \neg p, p, \neg p, \neg p, \dots \rangle$. We write "$\Sigma^\infty$" for the set of infinitely long strings on Σ.

The next three problems ask you to draw Büchi automata or Kripke structures. Be sure to clearly identify all initial and accepting states. Each initial state should have an unlabeled incoming edge from nowhere. Each accepting state should have a doubled border. (Kripke structures have initial states but do not have accepting states.) Use any drawing tools you like, but try to be neat. Do not make your answers any more complex than necessary.

In drawings with labeled edges, if you want to show a p edge and a $\neg p$ edge with identical endpoints, you may draw a single edge labeled *true*. This is just a notational shorthand; a single *true* edge really represents two distinct edges.

Problem 2.1 (5 points):
Draw a Büchi automaton that is equivalent to $\Diamond\Box p$. In other words, the automaton should accept exactly the strings in Σ^∞ that correspond to paths that satisfy $\Diamond\Box p$.

Problem 2.2 (5 points):
Draw a Büchi automaton that is equivalent to $\Box\Diamond p$.

Problem 2.3 (5 points):
$\Diamond\Box p$ and $\Box\Diamond p$ are not equivalent. Draw the Kripke structure for a model which satisfies one formula but not the other. State which formula your model satisfies and which it does not.

Problem 2.4 (5 points):
In general, what is the relationship between the set of models which satisfy $\Diamond\Box p$ and the set of models which satisfy $\Box\Diamond p$? Edge cases and boundary conditions matter, so when describing this relationship, either use precise mathematical notation or else be very careful and precise in your use of English prose. Briefly justify your claim.

Problem 2.5 (5 points):
Consider a unary temporal toggle operator, written "$\bowtie p$." A path satisfies $\bowtie p$ if and only if the truth value of p strictly alternates between successive states along the path. Note that we do not specify whether p is initially true or initially false, merely that it must alternate from each state to the next.

Give a translation from "$\bowtie p$" into the standard operators of linear temporal logic. This must be a simple, mechanical rewrite. Recursive definitions and extra, non-LTL notations are forbidden. You may use the standard temporal operators ($\Diamond$, $\Box$, $\bigcirc$) and the standard Boolean operators ($\wedge$, $\vee$, $\neg$) as in standard LTL. If you want to use other common Boolean operators, state how they are defined in terms of $\wedge$, $\vee$, and $\neg$.

3 Model Checking With Alloy

In Alloy, a *core instance* of a model is a set of atoms and relations that satisfy all of the facts of the model, whether given explicitly or implicit in signature declarations. If you ran Alloy using a trivial predicate that is always satisfied

```
pred alwaysTrue { }
run alwaysTrue for 8
```

then the core instances are all of the solutions Alloy might present using sets of size 8 or smaller. The next three problems ask you to build Alloy modules to model certain data structures. The idea is to create a model whose core instances are exactly the set of all valid data structures of the requested type. Your model should have just the right constraints and no more, so that it accepts all valid data structures and rejects all invalid ones.

You do not need to implement any operations on the data structures, such as search or insert or delete. Your task is merely to encode the invariants that represent valid instances of each data structure.

To facilitate testing, each problem includes a required API that your module *must* implement. This API includes one type signature and several functions that expose the relationships between parts of the data structure. You can implement these functions however you want, but their names, argument types, and result types must be exactly as given in the problem statement. Their implementations might be trivial or might perform complex calculations. That is up to you. You can add any other signatures, functions, predicates, etc. that you want. We don't care what you add, as long as you at least implement the required API.

For each problem, your module must also include **five** tests that demonstrate the correctness of your solution. Some tests should be paired **assert** and **check** statements that assert invariants which should hold, then check that there are no counterexamples to those invariants. Other tests should be paired **pred** and **run** statements that demonstrate

```
check NoMultipleRoots for 5
assert NoMultipleRoots {
    // tree never has more than one root
    lone api_root
}

run EmptyTree for 5
pred EmptyTree {
    // zero nodes is a legal tree
    no Node
}
```

Figure 1: Simple example tests

that your model does have some interesting core instances. You solution should yield no counterexample for each **assert**/**check**, and at least one valid example for each **pred**/**run**. See figure 1 for a few very simple examples of tests. Write your own tests: you will receive no points for re-submitting any of the tests from figure 1 as part of your solution.

You may use any of the standard utility models found in `models/util`, but this is not required. (Our reference solution, for example, does not use any of these utility models.) All other Alloy code must be written by you and you alone.

We will test your solutions using Alloy 4.2-rc as installed in `~cs706-1/public/alloy`. It is your responsibility to ensure that your solution works as intended in this version.

Problem 3.1 (15 points):
Build a model for binary trees in Alloy. Your Alloy module should enforce all appropriate invariants for a single binary tree. Do not implement any tree modification operations like insert and delete. Just provide the necessary signatures and facts to guarantee that every core instance of your model is a valid binary tree, and every valid binary tree is a core instance of your model.

You should already know what a binary tree is. If you do not, reasonable definitions can be found online at MathWorld, in Section 5.5.3 of the classic text *Introduction to Algorithms* by Cormen et al., or from a variety of other sources. If you think there is still some ambiguity, either ask for clarification or include comments in your solution that clearly state what the ambiguity is and how you resolved it.

Your module should provide all required API elements listed in figure 2 and **five** tests in any mix of **assert**/**check** and **pred**/**run**. When you submit your work, your solution to this problem should be in a file named `binaryTree.als`. The first non-comment line should read "**module** binaryTree" as in figure 2.

```
module binaryTree

// set of all nodes
sig Node { ... }

// tree root, or empty set if tree is empty
fun api_root: Node { ... }

// all (x, y) such that y is the left child of x
fun api_left: Node -> Node { ... }

// all (x, y) such that y is the right child of x
fun api_right: Node -> Node { ... }

// all (x, y) such that y is the parent of x
fun api_parent: Node -> Node { ... }

// all (x, y) such that y is somewhere in the subtree rooted at x
fun api_subtree: Node -> Node { ... }
```

Figure 2: Required API for binary tree model

Problem 3.2 (15 points):
Revise your solution for problem 3.1 to describe binary search trees on integers. A binary search tree on integers is a binary tree with the following additional properties:

1. Every node has an integer value.
2. The value of each node is strictly larger than the value of any node in its left subtree.
3. The value of each node is strictly smaller than the value of any node in its right subtree.

Note that because the second and third properties require *strictly* larger/smaller values, binary search trees (as we have defined them) cannot contain any duplicate values.

Your module should provide all required API elements listed in figure 3 and **five** tests not already used in the previous problem, in any mix of **assert**/**check** and **pred**/**run**. When you submit your work, your solution to this problem should be in a file named `searchTree.als`. The first non-comment line should read "**module** searchTree" as in figure 3.

```
module searchTree

// set of all nodes
sig Node { ... }

// tree root, or empty set if tree is empty
fun api_root: Node { ... }

// all (x, y) such that y is the left child of x
fun api_left: Node -> Node { ... }

// all (x, y) such that y is the right child of x
fun api_right: Node -> Node { ... }

// all (x, y) such that y is the parent of x
fun api_parent: Node -> Node { ... }

// all (x, y) such that y is somewhere in the subtree rooted at x
fun api_subtree: Node -> Node { ... }

// all (x, i) such that i is the value of x
fun api_value: Node -> Int { ... }
```

Figure 3: Required API for binary search tree model

Problem 3.3 (20 points):
Revise your solution for problem 3.2 to describe balanced AVL trees. A balanced AVL tree is a binary search tree with the following additional properties:

1. The *height* of any node is defined as the number of nodes on the longest path from that node to any of its descendants, including the node itself. For example, the longest path from a leaf to any of its descendants is just the leaf node itself, so leaf nodes must have height one. The height of an empty subtree is defined as zero.

   An equivalent definition is to say that the height of an empty subtree is zero, and the height of a node is one more than the maximum height of its children. For example, a leaf has two empty subtrees as children. Since those empty subtrees are defined to have height zero, the leaf must have height one.

2. The *balance* of any node is defined to be the height of its right child minus the height of its left child. Per the above definition for height, a missing child is treated as having height zero. When subtracting, do not swap the minuend and subtrahend!

3. All balanced AVL tree nodes have balances between -1 and +1, inclusive.

Figure 4 shows an example of an AVL tree with heights and balances for all nodes.

Your module should provide all required API elements listed in figure 5 and **five** tests not already used in previous problems, in any mix of **assert**/**check** and **pred**/**run**. When you submit your work, your solution to this problem should be in a file named `avlTree.als`. The first non-comment line should read "**module** avlTree" as in figure 5.

Note: Alloy treats integers as atoms, and therefore as unary singleton relations. For example, the Alloy expression "1" really represents the relation {(1)}. The "+" and "-" operators are relational union and relational difference, *not* arithmetic addition and subtraction. Thus, the Alloy expression "1 + 2" evaluates to {(1), (2)}, not 3! However, you may need to use integer addition and subtraction in part of your AVL tree model.

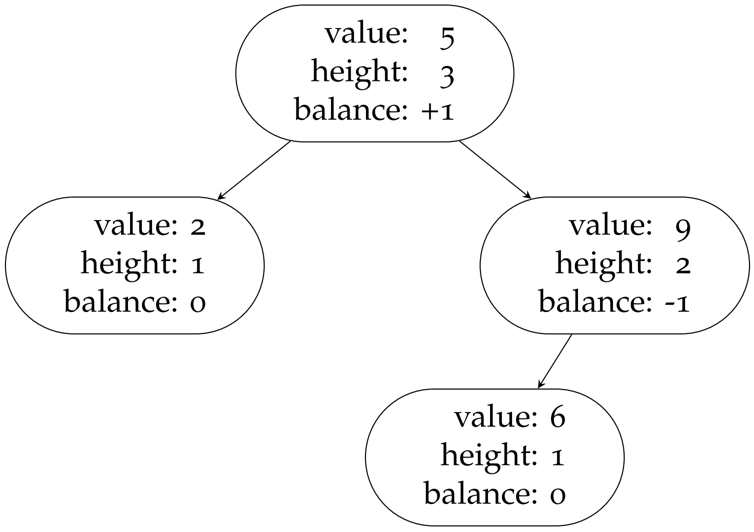

Figure 4: Example AVL tree with values, heights, and balances

Fortunately, Alloy does have some arithmetic routines you can use. Use "a.add[b]" or "add[a, b]" or "a.plus[b]" or "plus[a, b]" to add a and b. Use "a.sub[b]" or "sub[a, b]" or "a.minus[b]" or "minus[a, b]" to subtract b from a. See `~cs706-1/public/alloy/models/util/integer.als` for other useful arithmetic operations on integers; you are free to use anything from this module in your solution.

```
module avlTree

// set of all nodes
sig Node { ... }

// tree root, or empty set if tree is empty
fun api_root: Node { ... }

// all (x, y) such that y is the left child of x
fun api_left: Node -> Node { ... }

// all (x, y) such that y is the right child of x
fun api_right: Node -> Node { ... }

// all (x, y) such that y is the parent of x
fun api_parent: Node -> Node { ... }

// all (x, y) such that y is somewhere in the subtree rooted at x
fun api_subtree: Node -> Node { ... }

// all (x, i) such that i is the value of x
fun api_value: Node -> Int { ... }

// all (x, i) such that i is the height of the subtree rooted at x
fun api_height: Node -> Int { ... }

// all (x, i) such that i is the balance of the subtree rooted at x
fun api_balance: Node -> Int { ... }
```

Figure 5: Required API for AVL tree model