Modeling Coordinate Systems in Alloy

Ryan Jazayeri, Winston Wang, and Gregory Dennis

Abstract. We have investigated coordinate systems in Alloy. Coordinate structures are fundamental to many interesting systems, particular ones dependent on geography. Transportation systems, for instance, often adhere to grid formations. Our key case study for this research was the classic "Fifteen", or "Sliding Tiles," puzzle. We were able to use Alloy to solve the puzzle for arbitrarily sized boards. We also looked at other puzzles and games including Connect Four and Tic-Tac-Toe. We built a reusable matrix module to aid us in our modeling. We highlighted common patterns we found across our models.

The Matrix Model. The `std/matrix` model is a reusable library module we built to aid us in modeling specific coordinate systems. This model of a two-dimensional array bears strong similarity to the `std/seq` sequence model of a one-dimensional array. By observing how we extended the sequence model from one to two dimensions, we believe one could similarly build a three-dimensional model from our matrix model.

The matrix model contains a basic type named `Matrix`. `Matrix` is polymorphically parameterized over another type `t`, which when instantiated, becomes the type of the matrix's elements. The signature for Matrix contains a `matrixElems` field which is a quaternary relation from the Matrix to row indices to column indices to the type t. To clarify, each row index, column index pair, is mapped to a single element of the matrix. The matrix model uses to `std/ord` module to totally order the row and column index types.

The matrix model also contains a number of convenience functions that facilitate access to a matrix's elements. These include `MatrixAt`, which takes a row index and a column index as arguments, and returns the element located at the cross section of that row and column. The `MatrixRow` function, when given a row index as an argument, returns an `std/seq` sequence of the elements in that row. And `MatrixRight` returns the element to the right of the element it is given. Note that these last for functions, in addition to some other functions in the model, will not work properly if the matrix contains duplicate elements. That is, a function cannot really return the element b to the right of element a, if a appears twice in the matrix.

Having built a reusable matrix module for ease in modeling coordinate systems, we began to model three grid-based puzzles: Tic-Tac-Toe, Connect Four, and Fifteen.

Tic-Tac-Toe. For our first case study, we modeled the game tic-tac-toe. This model contains a singleton `Board` type, which has a field named `matrix` that points to a `Matrix` of `Squares`. Each `Square` has a `piece` relation that points to at most one atom of type `Piece`. `Piece` is partitioned into two disjoint singleton subsets `X` and `O`.

One of the key challenges in this problem is describing the tic-tac-toe itself. However with the help of the matrix model, the task was made a lot easier. We wrote a single function `tictactoe` that takes a piece *p* and a state *s* as arguments and is satisfied if there is a tic-tac-toe of *p* in state *s*.

Another important aspect of the Tic-Tac-Toe model is the function that describes a legal transition between states. It only constrains two aspects of the transition: (1) that the next piece to be played must not be the previous piece, i.e. an `X` must follow an `O` and vice-versa and (2) that the square the next piece is to occupy must currently be empty. A fact in our model constrains all consecutive states to abide by this legal transition.

Connect ~~Four~~ Three. For performance reasons, we chose to model "Connect Three" rather than "Connect Four." Through practice we found a four-by-four grid to be computationally infeasible for continual testing and analysis.

Our model for Connect Three is based on, and therefore very similar to, our model of tic-tac-toe. The key differences between the two are in the function that describes the winning state and in the function that constrains legal transitions. The function `threeInARow` describes the winning state for the Connect Three model. It accepts a `Piece` *p* and a `State` *s* as arguments and is true if *p* is forms a "connect three" in state *s*. Unlike the tic-tac-toe model, the function to constrain legal moves in Connect Four includes the additional stipulation that a piece must either be added on top of another piece or on the bottom row.

Fifteen. The Fifteen model was by far the most difficult of the three. The key basic type in this model is the `Tile`. The `Tile` type has the relations `atRow` and `atCol` to refer to the Tile's current row and column position in a given state, as well as `finalRow` and `finalCol` to refer to the location the tile must be located to solve the puzzle. `Tile` also has a relation `final` which points to another tile currently occupying the former tile's `finalRow` and `finalCol`. That is, a.final[s] = b means that in state s, b is located where a should be. This is primarily for visualization purposes—it enables us to easily see where each tile must be located for the puzzle to be solved. The board location holding no tile is modeled as holding the singleton `Empty` tile.

One fundamental difference between the Fifteen model and the Tic-Tac-Toe and Connect Four models is that the latter two include only a single static matrix of `Squares` that can in turn hold `Pieces`, whereas the Fifteen model anticipates a new `Matrix` of pieces per `State` and contains no notion of `Squares`. The matrix-per-state pattern was preferable in the Fifteen model because the matrix is always occupied by the same set of elements. This means the cardinality of `Tiles` is always equal to number of matrix positions. Adding squares and eliminating matrices in this case would have meant for a larger state space to search upon analysis. The exact opposite is true of the Tic-Tac-Toe and Connect Three models. Unlike Fifteen, they would have required twice as many pieces as matrix positions under the matrix-per-state pattern, making for a nearly-intractable state space.

Another interesting feature of the Fifteen model is the `legalMove` function that describes a legal transition between states. In Fifteen, each move can be thought of as swapping the `Empty` tile for a tile adjacent to it. This is essentially what the transition function stipulates: that the `Empty` tile move either up, down, left, or right, and that every other tile either stay in the same place, or swap places with the `Empty` tile.

When we began to run commands with large scopes on this model, we found the execution time to be simply too slow. To optimize performance, we introduced a new basic type `Move` and a new transition function `smartMove` that restricts each move from undoing the previous move. That is, an up move cannot be followed by a down move, nor a right move by a left move.

Analyses. With the models fully built, we were able to run some nice executions of game play for each of the puzzles. For tic-tac-toe we could ask it to show us an example in which `X` wins, in which `O` wins, in which `O` goes first and `X` Wins, in which they tie, etc. The same is true of Connect Four, except with Red and Black pieces. With the Fifteen model fully built, we were able to ask the analyzer to show us instances in which the game was solved in exactly 4, 5, 7, or any specific number of states. Some of these visualization are shown on the attached pages.