N-Queens Puzzle and Uppaal

Andy Gregoire
Department of Computing and Information Science, Kansas State University
e-mail: *amgregoi@ksu.edu*
Key Words: Uppaal, N-Queen, model, puzzle

Abstract

The N-Queens puzzle is an interesting challenge that has entertained many chess enthusiasts, this paper will give a brief overview of the Uppaal modeling, verification and validation tool as well as describe a model that verifies there is a solution to the N-Queens puzzle.

1. Introduction

The N-Queens puzzle is a derivative of a the classic Eight Queens puzzle, which is a problem of placing eight chess queens on an 8x8 chess board. The challenge of this puzzle is to arrange the queens in a manner in which no queen is attacking, or threatening, another queen. This means that a queen cannot be assigned to the same column, row or diagonal. The rules of puzzle are simple, now we go a step further, instead of finding a solution for an 8x8 chessboard, is it possible to find solutions for an NxN chessboard where N is in the set of all real numbers? This paper will give a brief overview of the Uppaal modeling platform as well as discusses the uppaal model that has been built in order to determine if a solution can be found for any number N.

2. Uppaal Platform

Uppaal is a computing platform to build modeling systems and validating and verifying real-time systems that have been modeled. Uppaal models act as state machines that can be controlled with logic and clocks in order to keep real-time validation conditions satisfactory. A system can have multiple models running inside of it. Each model inside the system is referred to as a template where the model is defined, and any logic is either stored in the templates local declaration file, or if something needs to be shared between all models it can be placed in the global declaration file. The two key components discussed below in regards to templates in Uppaal are locations and edges.

Locations

Locations act as a state in the models template, as you can see in Figure 1, it is represented by a circle and if it is the state in which the model starts in when it is initialized, it will contain a smaller circle inside of it to represent its initial location. Each location can be given a name to represent its location in the model. This is useful for validation in the Uppaal verifier by querying for example: “E<> User.School” which means, there exists a path where the model ‘User’ is on the location ‘School’ within the model.

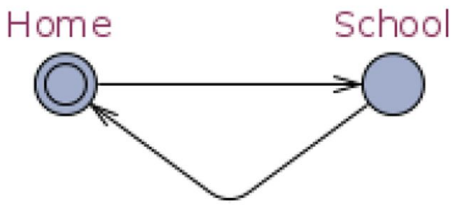

Figure 1 basic template with two locations, and two edges

There are two more types of locations, besides the initial location and default locations, which are Urgent, and Committed. When a model is in an urgent state clocks are in a sense paused because clocks are not allowed to increment. Committed locations also freeze time just like an Urgent location, however if a model arrives at a Committed location, the next action the system must take is a path, or edge, leading out of a committed location to simulate an atomic sequence of of operations

Edges

Edges are the connectors that start from one location to another. You can tell which location an edge starts at because it will not have an arrowhead on the side in which it starts, which can be seen in Figure 1. An edge is annotated with its selection, guard, synchronisation, and updates, each of these play a different role in how an edge is chosen, and what happens when you move along an edge to another location.

Selections

Selections are useful for assigning non deterministic values that may be used in an edges guard, synchronisation, and update. A simple example of a selections statement can be found in Figure 2 (value : int[0, 10]). What this statement is doing is assigning a value of type INT in the range 0 to 10 to our temporary identifier ‘value’. An important note when using selections, the identifier will temporarily replace any variables that share the same name.

Guards

Guards oversee whether an edge can be traversed or not at a given time, whether by the use of clocks, expressions, or functions that should evaluate to a boolean value. An edge can be made untraversable if the clock is currently set to 3, and the guard is made to be traversable when the clock is greater than 10. It is also possible to define a function in the local or global declarations to test some relevant information and determine if it is allowed to be traversed.

Synchronisations

Synchronisation is handled by channels, that are generally declared globally to be accessible for the whole system. As the name suggests, two processes (a process is an instantiated template) can be synchronised with each other if the guard on both edges is met and the synchronisation variable is labeled correctly. There are two ways to define a synchronisation label, with an exclamation point (!), or question mark (?) following the variable identifier, this can be seen in Figure 2. The bus! broadcasts on the channel bus and bus? receives the broadcast. If two processes are synchronised each edge will active at the same time in its respective template.

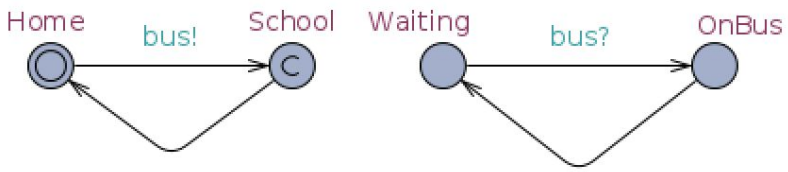

Figure 2 Example for how to set up synchronisation with channels

Updates

Updates, as the name suggests are how we make changes to the structures we declare either locally or globally. For example, if a template needs to keep records of how many steps it took to complete a cycle we could increment a counter in this step. This block is not limited to one update per edge, each update expression or method call is delimited by a comma.

Verifier

After building a template uppaal has a very useful verification tool built into it to test the models for various attributes. If it is important that a model is not suppose deadlock a simple query to run would be, E<> not deadlock, this tests that there exists a path that does not deadlock. This tool does have some limitations, just like in the previous case of testing deadlock, we cannot test all paths for not deadlock, A<> not deadlock, this is not valid. It is also useful for verifying that a model will eventually reach a certain state or location in your template, or testing your own declarations to be sure the model is behaving the way it is suppose to.

3. N-Queens Model

The puzzle discussed is for N consisting of any real number, however for the sake of building the model it handles real numbers from one to eight. The system contains a model that represents a chessboard, it

contains 64 locations. Each location represents one square on an 8x8 chessboard and each location has a connection to the initial location that is used to observe the state of the entire board.

Squares and their Edges

Each location representing a square section of a chessboard has two edge connected to it, the first comes from the initial observer location and one going back to the observer to see how the board has been manipulated. When leaving the observer to the square we make sure the edge is active by using a guard that calls a function, "getLocation". This method takes the coordinates of the square as parameters and verifies if it is a valid spot to place a queen or not, based on whether a queen is already there and if it would be in an attacking state to another queen. If both of these cases are valid, it returns true and guard allows the edge to activated. When the system follows the edge, an update function, "updateBoard", is called. It will also take the squares coordinates as parameters, and place the queen at this location. After the board is updated it returns back to the observer to determine if we are in a solution state, meaning we have succeeded. If the board has not reached a solution state and another queen can still be placed it will continue. However if there is a time when a queen cannot be placed and the board is not in a solution state the board setup is deemed impossible to find a solution. This means the pattern that was chosen did not meet the criteria of the puzzle.

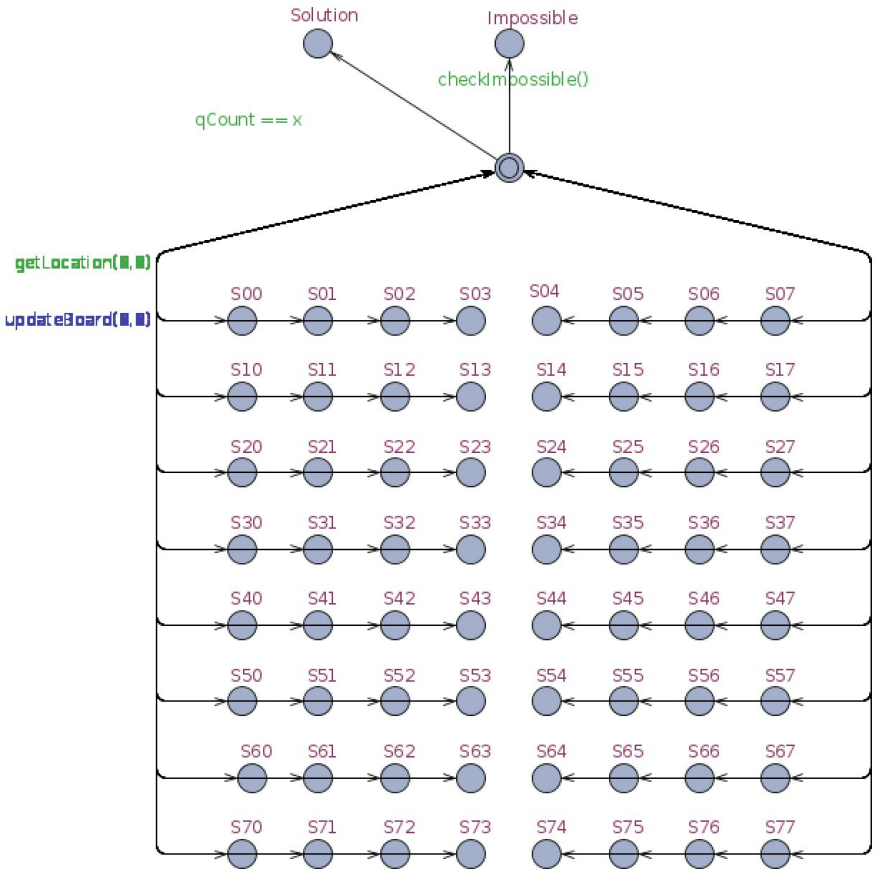

Figure 3 Model board for N-Queens puzzle

Solutions

Figure 3 is the template used to verify a solution exists to the N-Queen puzzle up to the size of 8. There are actually many solutions for each board size, and the number of solutions increases as the number of the queens and board size increases, in fact there are 12 unique solutions and 92 in total for an 8x8 chessboard. Because of how computationally expensive it is to find a solution, discovering all existing solutions of an NxN board takes an inordinate amount of time.

N-1 Solutions

While the number of board sizes of an NxN board is infinite, there do exist board sizes which do not contain any solutions. Specifically there two, 2x2 and 3x3 boards do not contain solutions that fit the puzzle requirements for the N-Queens puzzle. In the case of the 2x2 board, one queen puts the whole board in an unsafe state to place a second queen, and similar on a 3x3 board, after placing 2 queens the board is unsafe to place the third. So while these may not be solutions to the N-Queens puzzle, these are solutions to an N-1 Queens puzzle or finding the minimum number of queens on a board to make it unsafe.

4. Conclusion

We have given a brief overview of how to use the Uppaal model, verification and validation platform including locations and edges and the four key components that make connecting locations together with edges so useful. The N-Queens puzzle model was outlined to help reiterate some of these concepts while also solving an interesting problem. We found and verified that there do exist solutions for the N-Queens puzzle for all real numbers greater than one, with the exception of two and three. Although the model was only built to handle up to eight queens, it could be expanded to handle larger boards.