Puzzles for learning real-time model checking

Mitchell L. Neilsen
Department of Computer Science
Kansas State University
Manhattan, KS, USA

Abstract

This paper describes the utility of using puzzles to teach formal methods and real-time model checking. Puzzles allow us to teach model checking via finite games. The limitations of model checkers and the challenge of limiting the size of the state space are evident as students try to solve problems of varying size and difficulty. Students gain valuable practical experience in constructing models for puzzles before moving on to industry-sized problems.

Keywords: Cyber-physical systems, education, model checking, finite games, puzzles, real-time embedded systems.

1 Introduction

As real-time embedded systems become more complex, the role of formal methods to specify, design, implement, and validate such systems becomes even more important. A description of all formal methods is beyond the scope of this paper, but a nice introduction can be found on a NASA site: https://shemesh.larc.nasa.gov/fm/fm-what.html. The focus of this paper is to describe how model checking principles for real-time systems can be taught using finite puzzle games, and how model checking can be used to derive winning strategies for such games. In addition, model checkers can be validated using finite game test suites [8].

UPPAAL is a tool for validation (via graphical simulation) and verification (via automatic model checking) of real-time systems [4]. Models are constructed as a collection of timed automata; i.e., finite state machines with real-valued clocks.

Time is continuous and progresses globally at the same rate for the entire system. A system is composed of concurrent processes, modeled as communicating automaton [1]. Each automaton has a set of locations, and transitions can either delay or change location via an action transition. Action transitions can have a guard and synchronize with other automaton when fired. In UPPAAL, synchronization is achieved through hand shaking or rendezvous: two processes take a transition at the same time when one sends on a channel **a**, via **a!**, and the other process receives on channel **a**, via **a?**. Thus, a channel in UPPAAL is similar to a channel in SPIN, but with zero capacity. Model checking is basically an exhaustive search which covers all possible dynamic behaviors of the system. Most modern model checkers, including UPPAAL, use on-the-fly verification combined with symbolic techniques to reduce the verification problem to that of solving a simple constraint system [7, 12]. The verifier can be used to check for simple property invariants and a variety of different reachability properties. While most puzzles don't impose time constraints on game play, clocks can be used to derive novel solutions to puzzles as we shall see.

Section 2 presents puzzles that can be solved using UPPAAL. Section 3 presents puzzle solutions and some performance results. Finally, Section 4 concludes the paper and gives directions for future research.

2 Puzzle Problems

A number of simple puzzles and finite games have been used as examples for model checkers. In this section, we introduce a few new ones which have not been used for real-time model checking, to the best of our knowledge.

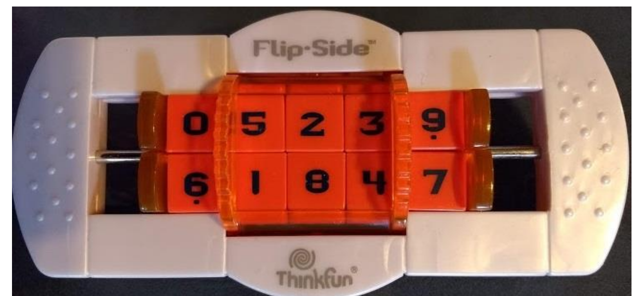


Figure 1. Flip-Side Puzzle

Flip-Side

The first puzzle, called "Flip-Side™", as shown in Figure 1, was invented by Ferdinand Lammertink. It is sold by ThinkFun. Each row of numbers can slide back and forth, and the middle three numbers can be flipped as shown in Figure 2, so that from Figure 1, "1,8,4" is now in the top row and "5,2,3" is in the bottom row.

Figure 2. Flip-Side Puzzle after Flip

The objective is to get the sequence 0,1,2,3,4 in the top row and 5,6,7,8,9 in the bottom row as shown in Figure 3.

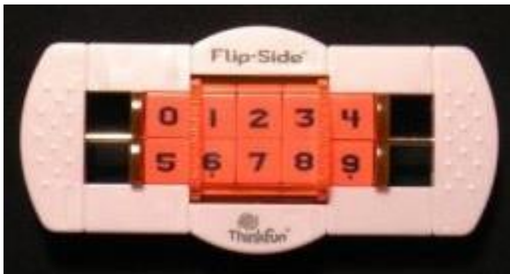


Figure 3. Flip-Side Puzzle Goal State

Flip-Side has been analyzed online on Jaap's Puzzle Page (https://www.jaapsch.net/puzzles/flipside.htm) by Jaap Scherphuis. With 10 digits, and enumerating all possible states, it is easy to see that there are at most 10! = 3,628,800 initial configurations with the numbers centralized, and it turns out that all of these initial configurations are solvable in at most 11 flips, but solutions may require many more moves where a move consists of shifting one of the rows left or right, or flipping the middle three. Only four initial configurations require 11 flips, and they are:

58069	**89672**	**27856**	**58469**
34127	**03514**	**03914**	**72301**

For this puzzle, it is feasible to enumerate all possible states, but for more complex problems, that may not be possible. For the puzzle in Figure 1, a shortest solution can be realized by the 19 moves: (1) top-right, (2) flip, (3) bottom-left, (4) flip, (5) bottom-right, (6) bottom-right, (7) flip, (8) top-left, (9) bottom-left, (10) bottom-left, (11) flip, (12) top-right, (13) flip, (14) top-left, (15) top-left, (16) flip, (17) bottom right, (18) flip, and (19) top-right. Thus, a total of 19 moves including 7 flips are required in solving this moderately difficult puzzle.

Triangular Tic-Tac-Toe

The second puzzle is Triangular Tic-Tac-Toe. It is played like regular tic-tac-toe such that X's or O's in any three in a row constitutes a win. Thus, if the triangle is numbered as shown in Figure 4, the winning sets are {{0,1,2}, {2,3,4}, {4,5,0}, {0,7,3}, {1,6,4}, {2,8,5}, {1,7,8}, {3,8,6}, {5,6,7}}. The board arrangement is from Martin Gardner's book Mathematical Circus.

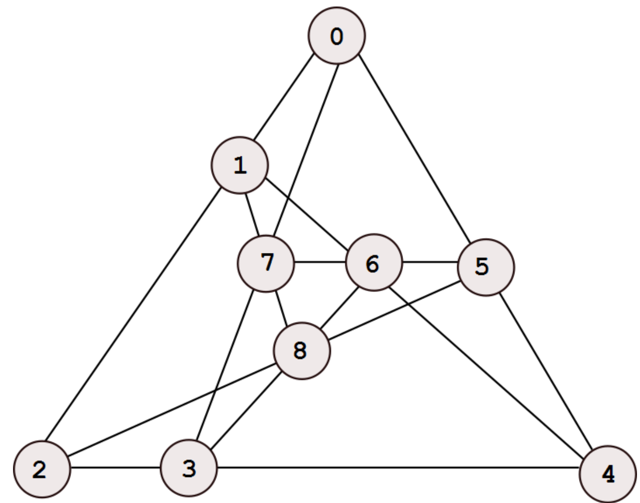


Figure 4. Triangular Tic-Tac-Toe

The interesting thing about this version of Tic-Tac-Toe is that the first player can always win, unlike regular 3x3 Tic-Tac-Toe where the game can always end in a draw. Consequently, this allows for an interesting assignment where we ask students to derive a "winning strategy" using a strategic model checker such as UPPAAL-Tiga [5]. Of course, it's not too much fun to play against the computer if it always wins whenever it plays first. Other questions to be solved relate to the minimum number of moves required to guarantee a win, or if a win is possible regardless of the first move.

Peg Solitaire

The third puzzle is called Peg Solitaire. This puzzle dates back to the 17th century. The game involves jumping over pegs on a board. The standard game fills the entire board with pegs except for one empty position. The objective is to empty the entire board by jumping and removing pegs by making valid moves, and in the end have only one peg remain. If the puzzle is solved so that the remaining peg is in the same position as the initially missing peg, then this is called a solution to the **complement** problem. A comprehensive coverage of all types of peg solitaire boards can be found online at George Bell's site: http://www.gibell.net/pegsolitaire/.

Rectangular boards where the number of rows ***n*** and the number of columns ***m*** are both even are called "even-even" boards. The 4x4 board starts with 15 pegs placed on the board. Since there are 15 pegs to start, all solutions require 14 **jumps**. When, the same peg jumps one or more pegs in succession, this is called a **move**. Solvable 4x4 boards require at least 9 moves. One possible 9 move solution is shown below in Figure 5. Jumps can only be made in a horizontal or vertical direction, diagonal jumps are not allowed. Consider the board shown below in Figure 5.

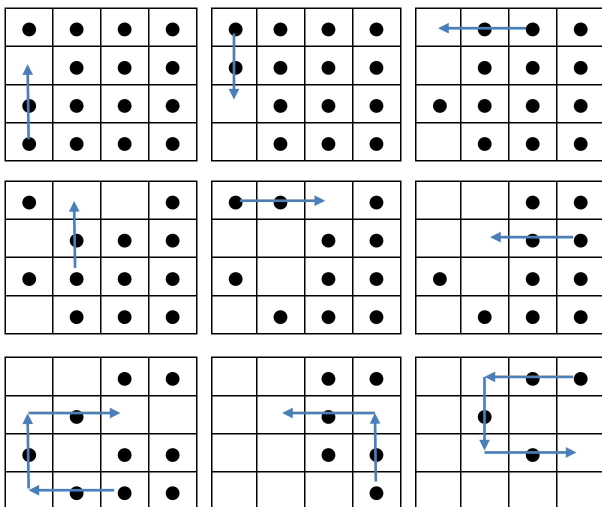


Figure 5. Peg Solitaire Solution

If we number the rows and columns 0 to 3 from the top-left corner, then the initial puzzle has a peg in all positions except for row 1, column 0, denoted as (1,0). The first move is to jump the peg in position (3,0) over the peg in position (2,0). The peg in position (2,0) is removed and the peg from (3,0) now resides in position (1,0). Note that the final move consists of starting with the peg in (0,3), jumping over pegs (0,2), (1,1), and (2,2) to end up in position (2,3) as shown. Thus, a total of 14 jumps and 9 moves are required to solve the puzzle. Model checking was used to determine the minimum number of moves required to solve a puzzle, and to determine the moves required as shown in Figure 5. Model checking can also be used to determine that all puzzles are solvable, but not in such a way that the final peg ends in the original empty slot; thus, there are no solutions to the complement problem on a 4x4 board.

Logi Toli and Modified Logi Toli

The final puzzle that we will consider is called Logi Toli as shown in Figure 6.

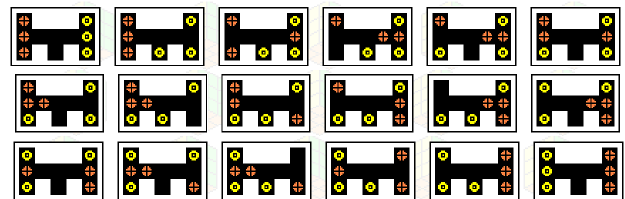


Figure 6. Logi Toli Puzzle and Solution [11]

The puzzle consists of tracks with six sliding pieces with three of each color on either end. The goal is to slide the pieces to exchange their positions so that the orange pieces have swapped positions with the yellow pieces. The minimum number of moves to solve the puzzle is 17 as shown in Figure 6. This is the same solution generated by the model checker.

Just for fun, we propose a variant of the Logi Toli puzzle that starts with 4 pieces on each side as shown below in Figure 7. The obvious problem to solve is to determine if the puzzle is still solvable; and if so, the minimum number of moves required.

The solution of this problem is left as an exercise for the interested reader.

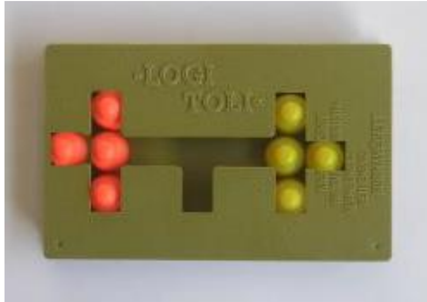


Figure 7. Modified Logi Toli Puzzle

3 Model Checking

After the problems are proposed, it is up to the students to derive concise models and properties that can be used to solve the problems. This section provides solutions to some of the problems and directions for further discovery.

To derive a model that doesn't suffer from the well-known "state-space explosion" problem, it is important to minimize the number of real-valued clocks included in the model because they highly influence the size of the state space. Other tricks are used as well, like using committed locations when possible and limiting the number, range, and scope of variables. Fortunately, UPPPAAL provides some convenient syntax which allows model builders to accomplish this goal efficiently.

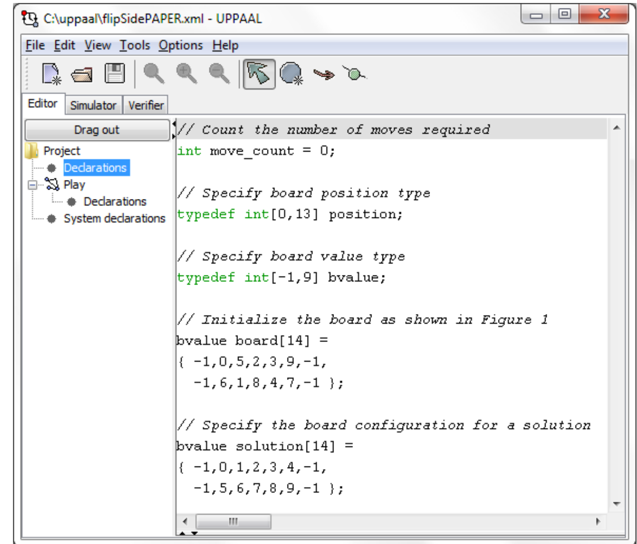
To limit the variable scope, students should favor the use of local variables over global variables, and minimize the use of channels. To limit range, they can use UPPAAL syntax to specify the range of values that a variable can hold; for example, in the peg solitaire model described below, just declare a new type called position, using:

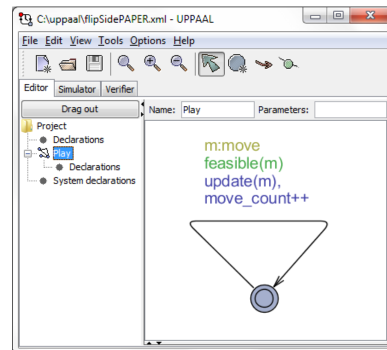
```
typedef int[0,3] position;
```

to declare that board positions (row or column) will be integeral in the range of 0 to 3.

Flip-Side

The model constructed for Flip-Side captures the board configuration and allowable moves. There are a total of 14 positions for the numbers 0-9 to occupy so the board can be modeled as a one-dimensional array of size 14. Letting a blank position be denoted as -1, we can specify the initial board configuration as shown in Figure 8 as an array containing board values{ -1,0,5,2,3,9,-1,-1,6,1,8,4,7,-1 }. Note the usage of typedefs to reduce the size of the state vector.





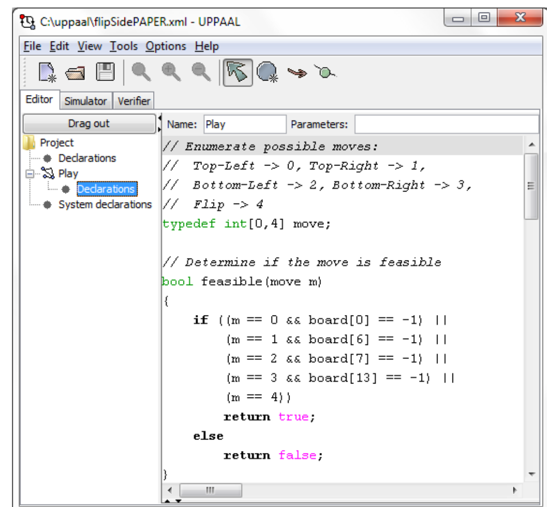


Figure 8. Flip-Side Model in UPPAAL

To model allowable game play, we can use a single automaton process called Play which has a single edge from the initial state to select a feasible move, update the board, and increment the number of moves made. Note that feasible(m) is a function that acts as a guard to ensure that the selected move can be made; update(m) is another function to update the board based on the move m selected.

Note that move is declared to be an integral type with values from 0 to 4, the moves are enumerated from 0 to 4. If the top-left position is empty then a top-left move (0) is feasible, and so on. A flip, denoted as move (4) is always feasible. To update the board after a move is a simple exercise. Finally, to verify that a solution can be obtained, use the query:

E<>(forall(i:position)(board[i]==solution[i]))

That is, on some path (E), is it eventually the case (<>) that for all positions, the board value is equal to the solution value. To find a solution that requires the fewest number of moves, we can request a diagnostic trace with the least number of transitions using the setting as shown in Figure 9. Once a trace is generated, we can go back to the simulator to step through the trace generated by the verifier.

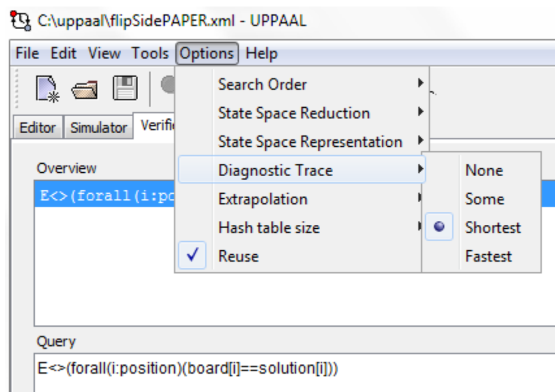


Figure 9. Options + Diagnostic Trace + Shortest

As you probably suspected, the solution generated is the same as the one given in Section 2 above. As a challenge solve one of the most difficult puzzles and determine the number of moves required.

Triangular Tic-Tac-Toe

Triangular Tic-Tac-Toe is a two-player game that can be modeled as a timed two-player game using UPPAAL-Tiga [5]. Control transitions are denoted as solid edges, and uncontrolled transitions, typically denoting the environment, are denoted as dashed edges. To constrain the state space size, the possible moves can be denoted as an integer from 0 to 8 by using the enumeration of the board shown above. The set of all winning combinations can be store in a two-dimensional array. Finally, a real-valued clock c is needed to force progress as shown in Figure 10.

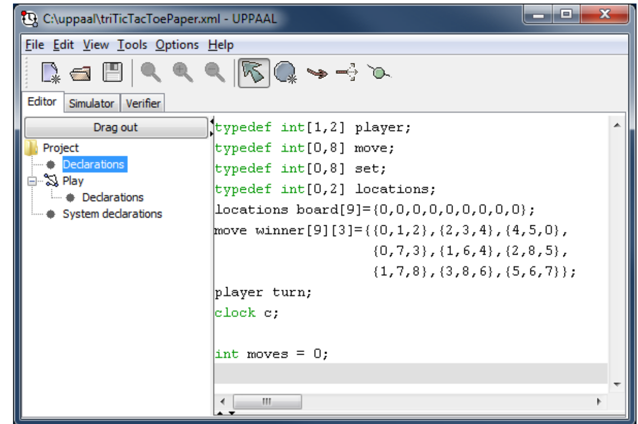


Figure 10. Global declarations

A single process, Play, shown in Figure 11, can be used to model the system.

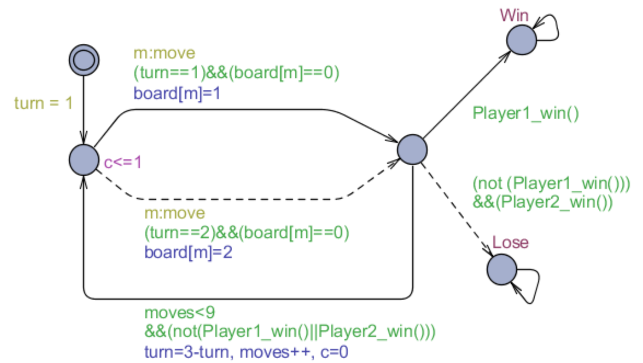


Figure 11. Play automaton

From the initial state, the controlled player, Player 1, is set to make the first move by setting turn = 1. To verify that the first player can always win, we can just use the property that under control, it is always the case on all paths, we reach the Win state where Player 1 wins as shown in Figures 11 and 12.

If Player 1 is not allowed to make the first move, then the property is no longer satisfied; to check, just change “turn = 1” to “turn = 2”. It is interesting to note that without the addition of the real-valued

clock to force progress, the propery is also not satisfied. In this case, the environment, Player 2, can just delay, refusing to make a move, when Player 1 has forced them into a corner.

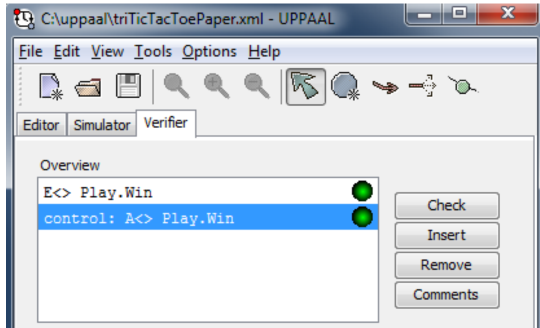
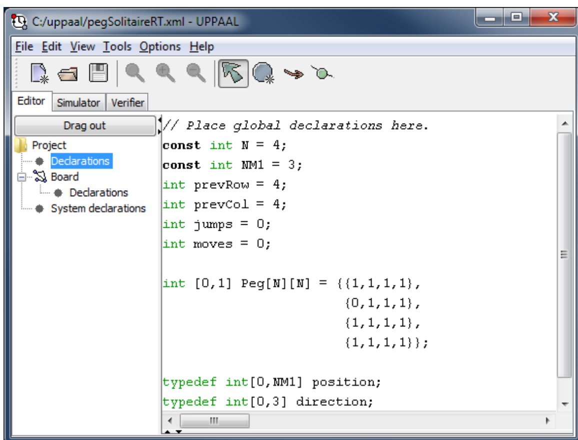


Figure 12. UPPAAL-Tiga verifier

Peg Solitaire

Peg solitaire can be solved using a carefully constructed UPPAAL model. For the 4x4 problem shown in Figure 5, the following model can be used.



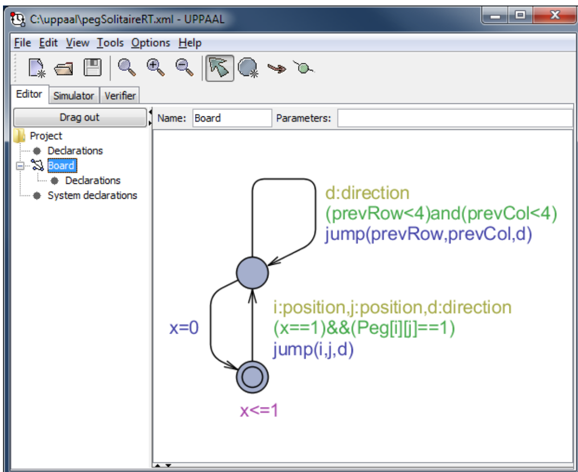


Figure 13. Peg solitaire model

From the initial state, select a random peg at location row i, column j, and try to move in direction d, by calling the jump function shown in Figure 14, only the first part of the function is shown, but the other parts are similar. The real-valued clock or the moves counter can be used to determine the minimum number of moves required to solve the problem. Recall that at least 9 moves are required, if the same peg jumps several other pegs, it only counts as a single move.

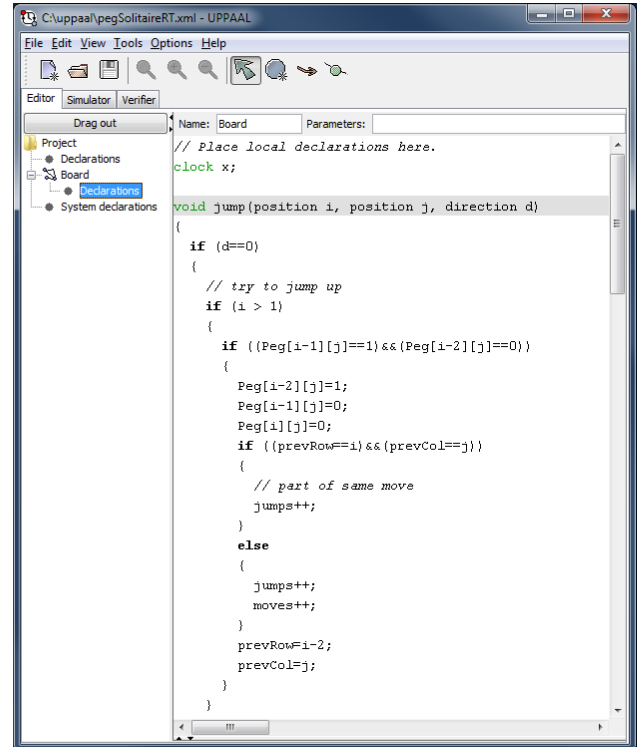


Figure 14. Jump function fragment

The same results are found using the verifier.

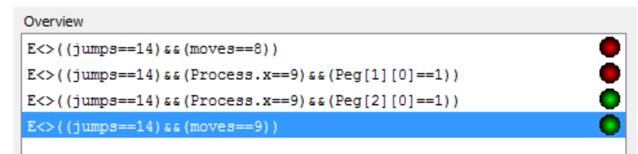


Figure 15. Verification results

To generate a trace, just select Options + Diagnostic Trace + Fastest -- note that Shortest will not work because all solutions involve exactly 14 jumps or transitions, but by using the real-valued clock, only the first jump in each move takes 1 time unit, and the

transitions to make additional jumps in the same move (the top transition in Figure 13) take no time.

Logi Toli and Modified Logi Toli

Due to space limitations, the Logi Toli puzzle and the Modified Logi Toli puzzles are left as exercises for the interested reader. However, we would be happy to supply anyone with complete solutions upon request. Both have solutions.

Once students master solving puzzle problems and learn model checking "tricks of the trade", they are ready to solve more challenging industry-sized problems, and derive models for systems that aren't as obvious, such as real-time seed counters [9,10] or real-time communication systems [13]. Qualitative assessment has shown that the use of puzzles is an effective way to motivate students to learn how to use model checkers effectively.

4 Conclusions

Overall, the use of puzzles has been an effective approach to teach real-time model checking. This paper presents several novel puzzle problems that lend themselves to real-time model checking, and solutions to some of the problems are provided to demonstrate techniques that can be used to build industry-sized models, and as a side also verify puzzle properties.

Acknowledgements

This material is based upon work supported by the National Science Foundation under NSF-IOS Grant No. 1543958. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Alur and D. Dill, "A theory for timed automata", In Theoretical Computer Science, Vol. 125, pp. 183–235, 1994.

[2] C. Baier and J.P. Katoen, "Principles of Model Checking", MIT Press, Cambridge, MA, 2008.

[3] J.K. Barker and R. Korf, "Solving peg solitaire with bidirectional BFIDA*", in Proceedings of the 26th AAAI Conference on Artificial Intelligence, pp. 420-426, 2012.

[4] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems", In Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, Oct. 22-24, 1995.

[5] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, D. Lime, "UPPAAL-Tiga: Timed games for everyone", In L. Aceto and A. Ingolfdottir (eds.) Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT 2006), Reykjavik University, 2006.

[6] E.M. Clarke, E.A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging", Communications of the ACM, 52(11):74–84, 2009.

[7] K.G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems", In Proc. of Fundamentals of Computation Theory, Vol. 965 of Lecture Notes in Computer Science, pp. 62–88, 1995.

[8] M.L. Neilsen, D.H. Lenhert, M. Mizuno, G. Singh, J. Staver, N. Zhang, K. Kramer, W.J. Rust, Q. Stoll, M.S. Uddin, "Encouraging interest in engineering through embedded system design", In American Society of Engineering Educators (ASEE) Computers in Education Journal, Vol. XV, No. 3, pp. 68-77, July 2005.

[9] M.L. Neilsen, S.D. Gangadhara, and S. Amaravadi, "Extending watershed segmentation algorithms for high-throughput phenotyping on mobile devices", in Proc. of the 30th International Conference on Computer Applications in Industry and Engineering, San Diego, CA, 2017.

[10] M.L. Neilsen, C. Courtney, S. Amaravadi, Z. Xiong, J. Poland and T. Rife, "A dynamic, real-time algorithm for seed counting", in Proc. Of the 26th International Conference on Software Engineering and Data Engineering, 2017.

[11] J. Scherphuis "Jaap's Puzzle Page", retrieved from https://www.jaapsch.net/puzzles/logitoli.htm, 2018.

[12] N.V. Shilov and K. Yi, "Puzzles for learning model checking, model checking for programming puzzles, puzzles for testing model checkers", In Electronic Notes in Theoretical Computer Science 43, 2001, http://www.elsevier.nl/locate/entcs/volume43.html.

[13] Y. Wang, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving", In Proc. of the 7th International Conference on Formal Description Techniques, 1994.