Two-player games 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 two-player games to teach formal methods and real-time model checking. Many systems that we want to model involve two parts: a discrete control program and a continuous environment. One player can be used to play the role of the control program and the other can play the role of the environment. Consequently, two-player games allow us to teach model checking techniques that can also be applied to build models to solve complex problems found in industry. 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.

Keywords: Cyber-physical systems, embedded systems, model checking, finite games, real-time 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 two-player 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. To model two-player games, a strategic model checker which extends UPPAAL, called UPPAAL-TIGA [5], can be used. Controlled edges can be used to model moves by one player and uncontrolled edges can be used to model moves by the other player. Finally, while most games don't impose time constraints on game play, clocks can be used to derive novel solutions to games and force progress, as we shall see.

Section 2 presents some two-player games that can be solved using UPPAAL-TIGA. Section 3 presents model solutions and some performance results. Finally, Section 4 concludes the paper and gives directions for additional research.

2 Two-Player Games

Several simple logic puzzles and multi-player games have been used as examples for model checkers [14]. In this section, we introduce a few new ones which have not been used by others for real-time model checking, to the best of our knowledge. Then we develop simple models to develop strategies to play games using real-time model checkers. We'll start by describing some twists on a traditional two-player game, tic-tac-toe.

Wild Tic-Tac-Toe

The game of Tic-Tac-Toe is traditionally played by two players with paper and pencil on a 3x3 grid. The players take turns marking X's and O's in any open position. The first player able to place three of their marks in a horizontal, vertical, or diagonal row wins the game. For the board shown in Figure 1, the winning sets are { {0,1,2}, {3,4,5}, {6,7,8}, {0,3,6}, {1,4,7}, {2,5,8}, {0,4,8}, {2,4,6} }. In the game play shown in Figure 1, O wins with marks in positions 6, 7, and 8. Unfortunately, players soon discover that optimal play always leads to a draw in which neither player wins.

0	**1**	**2**
3	**4**	**5**
6	**7**	**8**

X	**O**	**X**
O	**X**	**X**
~~**O**~~	~~**O**~~	~~**O**~~

Figure 1. Tic-Tac-Toe Board and Game

One variant, called *Wild Tic-Tac-Toe*, allows a player to play either X or O on their move. If a player can win the game with either an X or an O played, then they are declared the winner. In this case, the first player can always win the game, but it requires a bit more reasoning to develop a winning strategy. Also, it's easier for a player to make an erroneous move. Another interesting variant of tic-tac-toe is played on a triangular board as described below.

Triangular Tic-Tac-Toe

The second game, called Triangular Tic-Tac-Toe, is played like regular tic-tac-toe such that X's or O's in any three in a row constitutes a win, but the rows are organized into a triangle. Thus, if the triangle is numbered as shown in Figure 2, 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.

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.

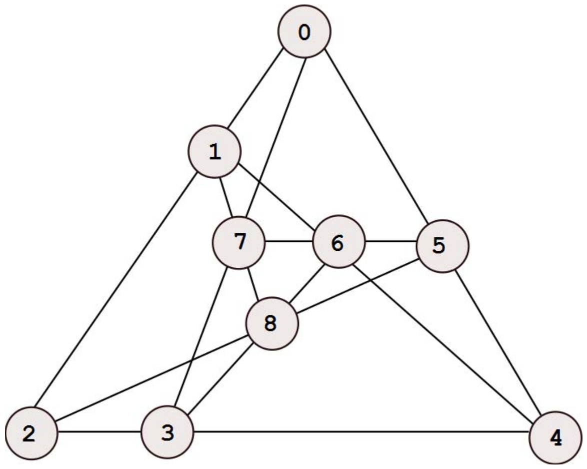


Figure 2. Triangular Tic-Tac-Toe

Hot Spot Tic-Tac-Toe

The final game is a brand-new variant of tic-tac-toe that makes for a more interesting game which doesn't always result in a win or a draw with optimal play. Note that the triangular tic-tac-toe game can always be won by the first player and has a total of 9 winning combinations, but the regular tic-tac-toe game can always end in a draw and only has a total of 8 winning combinations. The unique idea behind Hot Spot Tic-Tac-Toe is to randomly select three spots or positions on the board to also constitute a winning combination, thus giving a total of 9 winning combinations. Of course, if the hot spot winning combination is the same as an existing combination, then the game will end in a draw with optimal play. But if the game randomly selects the hot spot combination {1,3,5}, then the first player can always win. In this way, the game won't always end in a draw, and the play strategy changes based on the random hot spot numbers selected.

0	**1**	**2**
3	**4**	**5**
6	**7**	**8**

Figure 3. Hot Spot Tic-Tac-Toe Board

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.

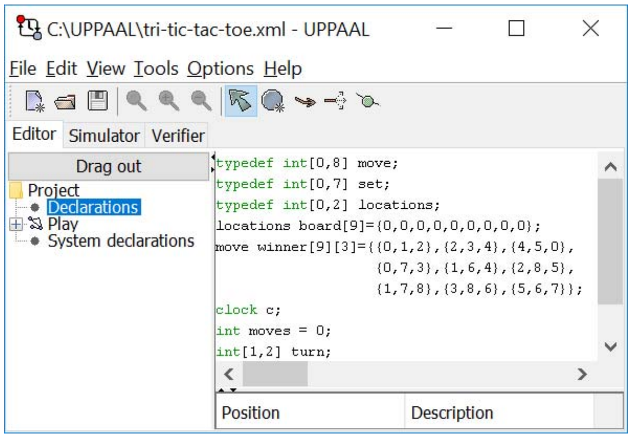
To limit the variable scope, students should favor the use of local variables over global variables, and they should also minimize the use of channels. To limit range, they can use UPPAAL syntax to specify the range of values that a variable can hold; for example, in the model for triangular tic-tac-toe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

```
typedef int[0,8] move;
```

to declare that move positions will be integers in the range of 0 to 8 as shown in Figure 4.

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 in Figure 2. Each board position can be enumerated as blank (0), a play of X (1), or a play of O (2). The board is initialized to all blanks. Then, the set of all winning combinations can be stored in a two-dimensional array, called winner; e.g., if board positions 0, 1, and 2 are all the same value (1 or 2), then a winning state has been realized. Finally, a real-valued clock c is needed to force progress as discussed below.



```
typedef int[0,8] move;
typedef int[0,7] set;
typedef int[0,2] locations;
locations board[9]={0,0,0,0,0,0,0,0,0};
move winner[9][3]={{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}};

clock c;
int moves = 0;
int[1,2] turn;
```

Figure 4. Global declarations

A single process, Play, shown in Figure 5, can be used to model the system. 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 eventually reach the Win state where Player 1 wins as shown in Figures 5 and 6.

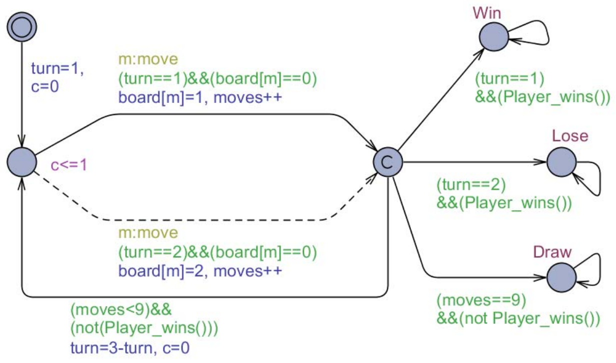
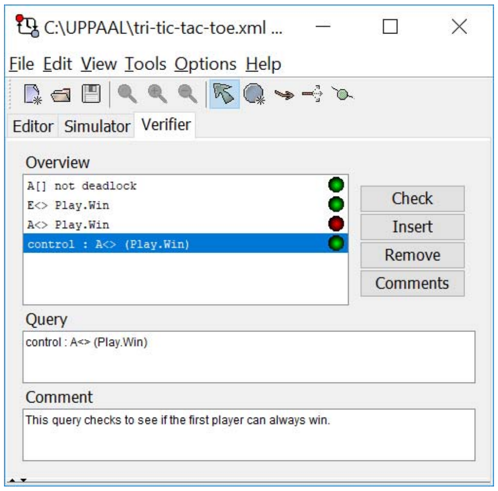


Figure 5. Play automaton

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" in the transition from the initial state. 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 Player 2 into a corner without a move that will prevent Player 1 from winning.


C:\UPPAAL\tri-tic-tac-toe.xml ...
File Edit View Tools Options Help
Editor Simulator Verifier
Check
Insert
Remove
Comments

Overview

```
A[] not deadlock          ●
E<> Play.Win              ●
A<> Play.Win              ●
control : A<> (Play.Win)  ●
```

Query

control : A<> (Play.Win)

Comment

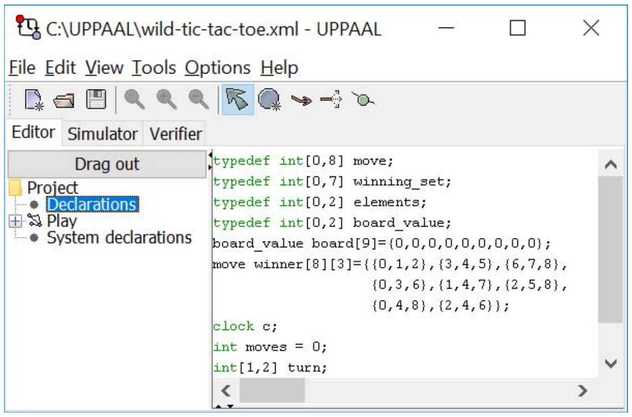
This query checks to see if the first player can always win.

Figure 6. UPPAAL-TIGA verifier

As shown in Figure 6, we can also verify that the system never deadlocks and that without control, it's no longer the case that the first player will always eventually win; i.e., (A<> Play.Win) is not satisfied.

Wild Tic-Tac-Toe

In *Wild Tic-Tac-Toe*, players can win the game by either completing the board with three X's or three O's, so the strategy is much different than regular tic-tac-toe or triangular tic-tac-toe. The play becomes much more defensive to ensure that the opponent is not able to win on their next move, while at the same time forcing the opponent to make a move that allows you to win.


C:\UPPAAL\wild-tic-tac-toe.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Drag out
Project
Declarations
Play
System declarations

```
typedef int[0,8] move;
typedef int[0,7] winning_set;
typedef int[0,2] elements;
typedef int[0,2] board_value;
board_value board[9]={0,0,0,0,0,0,0,0,0};
move winner[8][3]={{0,1,2},{3,4,5},{6,7,8},
                   {0,3,6},{1,4,7},{2,5,8},
                   {0,4,8},{2,4,6}};

clock c;
int moves = 0;
int[1,2] turn;
```

Figure 7. Global declarations

A model, similar to the one in Figure 5, can be used, but the move made can either set a given board position to 1 or 2. Also, the set of winning sets are the same as in regular tic-tac-toe, so global declarations are updated as shown in Figure 7. The updated play automaton is shown below in Figure 8. In this case a play, p, is either 1 or 2.

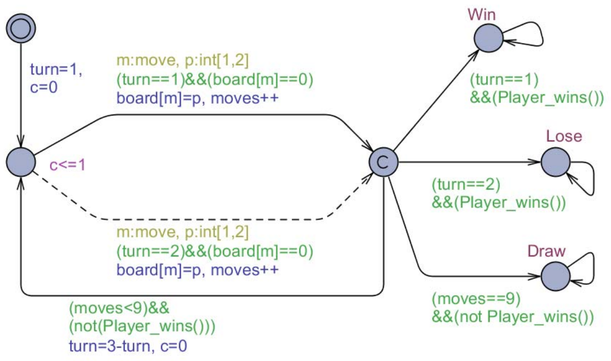


Figure 8. Play automaton

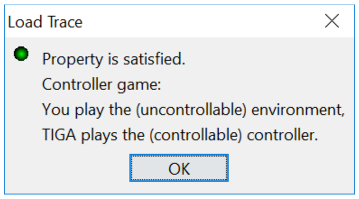
As in triangular tic-tac-toe, the first player can always win. But some of the plays are interesting. The strategy found by UPPAAL-TIGA is to play X in the center. If the opponent plays O in a corner, then TIGA counters to play O in the opposite corner. Likewise, if the opponent plays O in a side position (positions 1,3,5, or 7), TIGA counters by playing an O in the opposite side. In either case, the opponent is forced to make a move which causes them to lose.

The function to test if a player wins checks to see if there exists a winning set of board positions all set equal to 1 or 2:

```
bool Player_wins() {
  if (exists(i:winning_set)
    exists(p:int[1,2])
      forall(j:elements)board[winner[i][j]]==p)
        return true;
  else
    return false;
}
```

The same properties shown in Figure 6 for triangular tic-tac-toe are satisfied. In particular, on some path (E), eventually (<>) it is possible for the first player to win, and further under control on all paths (A), eventually (<>) the first player can win.

To enable play against the strategy generated, just select Options + Diagnostic Trace + tiga_some. When the property is checked, the dialog shown in Figure 9 is displayed. After generating a winning strategy, you can make moves for the environment, and TIGA counters with controllable moves made by the controller as shown in Figure 10. This allows students to observe the strategy first-hand by making plays against the computer.

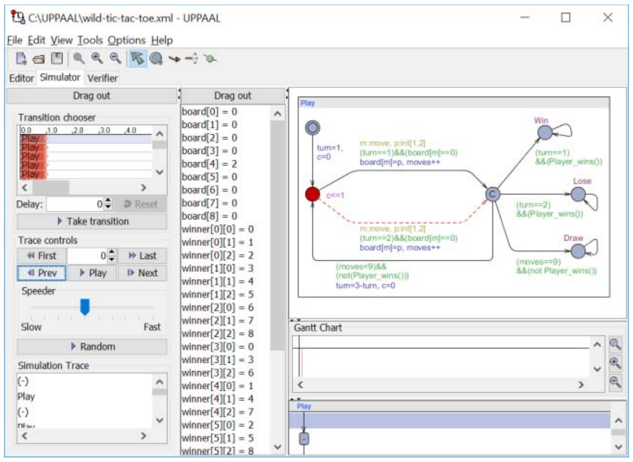

Load Trace

Property is satisfied.
Controller game:
You play the (uncontrollable) environment,
TIGA plays the (controllable) controller.

OK

Figure 9. Load trace dialog

The controller starts by setting a 2 in the center. If the environment counters by playing a 1 in a corner, the controller counters by playing a 1 in the opposite corner, forcing the environment to lose.


C:\UPPAAL\wild-tic-tac-toe.xml - UPPAAL

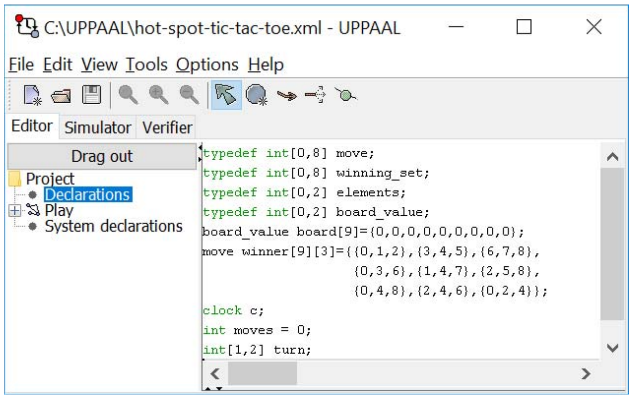
```
board[0] = 0
board[1] = 0
board[2] = 0
board[3] = 0
board[4] = 2
board[5] = 0
board[6] = 0
board[7] = 0
board[8] = 0
winner[0][0] = 0
winner[0][1] = 1
winner[0][2] = 2
winner[1][0] = 3
winner[1][1] = 4
winner[1][2] = 5
winner[2][0] = 6
winner[2][1] = 7
winner[2][2] = 8
winner[3][0] = 0
winner[3][1] = 3
winner[3][2] = 6
winner[4][0] = 1
winner[4][1] = 4
winner[4][2] = 7
winner[5][0] = 2
winner[5][1] = 5
winner[5][2] = 8
```

Figure 10. Play against the controller

An obvious follow-on question is what happens if we play wild tic-tac-toe on a triangular tic-tac-toe board. Counter to our intuition, the combined game is not so advantageous to the first player, and indeed the first player is not guaranteed to win.

Hot Spot Tic-Tac-Toe

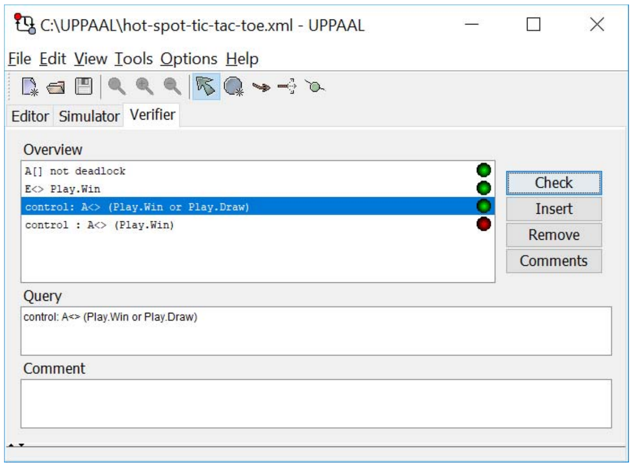
The final game to consider is a new game that we just invented and call Hot Spot Tic-Tac-Toe. Before play, three random numbers, call hot spots, are set by the game. They could be set to be different than any existing winning combination. Play proceeds like regular tic-tac-toe, but the existing standard set of winning combinations are augmented with the set of hot spots. Depending on the hot spots, the first player may be able to win with optimal play. For example, if the hot spots are 1, 3, and 5 as shown in Figure 3, then the first player can always eventually win.


C:\UPPAAL\hot-spot-tic-tac-toe.xml - UPPAAL

```
typedef int[0,8] move;
typedef int[0,8] winning_set;
typedef int[0,2] elements;
typedef int[0,2] board_value;
board_value board[9]={0,0,0,0,0,0,0,0,0};
move winner[9][3]={{0,1,2},{3,4,5},{6,7,8},
                   {0,3,6},{1,4,7},{2,5,8},
                   {0,4,8},{2,4,6},{0,2,4}};
clock c;
int moves = 0;
int[1,2] turn;
```

Figure 11. Hot spots {0,2,4}

However, if the hot spots are 0, 2, and 4, as shown in Figure 11, then the first player cannot always eventually win. So, it really depends on which hot spots are selected by the game. Most hot spot sets of size 3 give the first player an edge, but any hot spot set that includes 4 is not a winning configuration for the first player, but the first player can keep from losing as shown in Figure 12.


C:\UPPAAL\hot-spot-tic-tac-toe.xml - UPPAAL

Overview

```
A[] not deadlock
E<> Play.Win
control: A<> (Play.Win or Play.Draw)
control : A<> (Play.Win)
```

Query

control: A<> (Play.Win or Play.Draw)

Comment

Figure 12. Verify first player can win or draw

If we allow the hot spot set to contain a different number of positions, then not surprisingly, smaller sets are more likely to allow the first player to win. All sets of size 1 would be instant winners for the first player. However, not all sets of size 2 are so lucky; for example, the set {0, 4} is one case.

At the other extreme, some sets of size 4 are winning hot spot sets for the first player, including the sets {0, 2, 3, 5} and {0, 2, 3, 8}, and set symmetrical to these sets. There aren't very many sets of size 4 that result in a winning configuration for the first player.

4 Conclusions

While solving puzzle problems and constructing models for two-player games, students master model checking "tricks of the trade". Then, 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 two-player games is an effective way to motivate students to learn how to use model checkers effectively.

Overall, the use of two-player games has been an effective approach to teach real-time model checking. This paper presents several novel two-player game 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 game properties.

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, S. Amaravadi, "Extending watershed segmentation algorithms for high-throughput phenotyping on mobile devices", in Proc. of the 30th International Conference on Computer Apps 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", retr. 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.

[14] M.L. Neilsen, "Puzzles for learning real-time model checking", In Proc. of the International Conference on Frontiers in Education, FECS 2018, pp. 11-17, 2018.