Teaching Model Checking via Games and Puzzles

Bernd-Holger Schlingloff [1,2]

[1] Humboldt-Universität zu Berlin
[2] Fraunhofer FOKUS, Berlin

Abstract. Puzzles and games give a strong motivation for humans to deal with formal objects: people spend hours and hours in seemingly useless board games, moving pebbles or cards according to prescribed rules, trying to beat their opponent in a game or just solving a puzzle. In this position paper we show how to use this human obsession in order to teach students formal methods, in particular, SAT solving and model checking.

Keywords: Model checking · Puzzles · Games · Formal Methods · Education

1 Introduction

From an evolutionary point of view, being able to plan ahead and foresee a sequence of events is an essential survival skill. By considering different alternatives and their outcome, thinking beings are able to accomplish complex tasks and outperform competitors. Games and puzzles are mental exercises which train and improve the planning capacities of the human brain. Winning a game or solving a puzzle triggers a neural reward system, which reinforces the interest in the game or in similar puzzles. Therefore, humankind has always been interested in games. The earliest board games have been dated back to 2600 BC, with the oldest written rule set recorded in 177 BC [BrMus]. Being able to play board games like Chess and Go has been a criterion for machine intelligence at least since 1950, when Turing suggested a simple chess problem as a question in his imitation game [Tur 50]. Currently, the best computer programs for these games play better than any human opponent [SHS+ 17].

We want to demonstrate how the interest in games and puzzles can be used in a graduate course on automated verification, on advanced bachelor's or master's level (3^{rd} to 5^{th} year in computer science). We suggest to represent widely-known challenges as input for verification tools, and to have the students explore the capabilities and limitations of the tools via these examples. For specific model checkers and challenges, this suggestion has been made before (see, e.g., [SY01], [vDR07] and [EJV05] for a certain coin game, an epistemic riddle, and an elevator problem). Here, we claim that this approach can be used systematically to introduce various modelling formalisms (propositional logic, labelled transition systems and concurrent game structures) as well as specification languages (temporal and strategic logics). The proposed method has been tried in the classroom by the author; however, this paper is not an “experience report”. It rather gives concrete suggestions how to teach modelling and model checking on the graduate level in an interesting and intriguing way. Of course, to avoid the impression that formal methods have been invented to solve games, the playful examples presented here should be accompanied by appropriate examples from different industrial domains (which are, however, outside the scope of this paper).

The paper is structured as follows. Section 2 contains some general remarks of modelling puzzles and games with states and transitions. In Section 3, we model the well-known Sudoku puzzle with propositional logic and show how to use SAT-solving to find a solution. Section 4 introduces labelled transition systems for modelling and solving solitaire puzzles such as crossing a river and sliding blocks. In Section 5, we extend our teaching approach to concurrent game structures for modelling board games such as Tic-tac-toe. As a more scalable example, we introduce Tourality, which is a competitive rally game for two or more players. Finally, in Section 6 we conclude with ideas how to extend the approach to SMT modelling, discrete Markov chains, and epistemic structures.

2 Modelling puzzles and games

The notion of a state-transition system is fundamental for many formal methods. State-transition systems are the basis for finite automata, Kripke structures, labelled transition systems, and many other formalisms. Basically, a state-transition system is a graph consisting of nodes (i.e., states) and edges (i.e., transitions), where each edge connects two nodes. Alternatively, the set of transitions can be introduced as a relation between states. But what is a state? In the most general setting, a state of a system is defined to be a complete description of the system, consisting of the value of all parameters that determine the properties of the system. For a computer program, its state consists of the values of all variables or memory locations at some point in time. During a computation, a computer program passes through several states, by assigning values to variables. The verification task often amounts to showing that certain states are (un-)reachable. For example, we might want to show that a certain error state never occurs during execution, or that every computation leads to a state where the correct result has been calculated. For distributed systems, we might want to show that the interaction of the programs behaves correctly, i.e., that either no unwanted global state or some desired system goal state is reached, or that the system behaves correctly even in the presence of an intruder.

Many logical puzzles and games can be described as state-transitions systems. Puzzle or game elements are jigsaw pieces, letters, cards, pebbles, pegs, dice, etc. The state variables describe the current distribution of elements on the table or amongst the players. The game ends or the puzzle is solved, if a certain configuration is reached. In a multi-player game, usually the game ends if one of the player has reached a winning state.

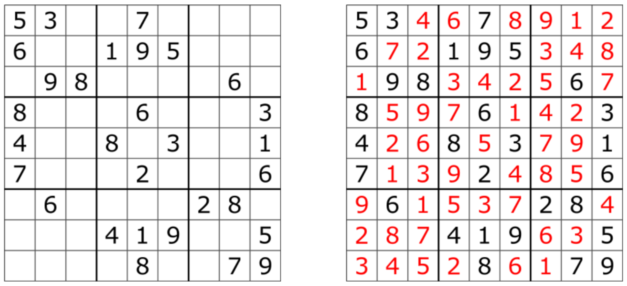
Thus, both computations and games can be modelled in terms of states and transitions. This viewpoint is useful to convey the ideas underlying formal verification methods and model checking to graduate students. Subsequently, we show how to teach (some parts of) program verification and formal methods via puzzles and games.

3 Combinatorial puzzles and SAT solving

SAT solving is a technique which can be readily used to solve a certain class of combinatorial puzzles. We demonstrate this with the well-known example of Sudoku. This puzzle is given on a square board of 9×9 fields, which is divided into nine 3×3 sub-squares. Some of these fields are marked with a digit from 1 to 9, some are empty. The challenge is to fill the empty fields with digits (1 to 9) such that

1. each row of the board contains each digit exactly once,
2. each column of the board contains each digit exactly once, and
3. each sub-square contains each digit exactly once.

For any given marking of the field, there may exist no, exactly one, or more than one solution. Usually, the given marking is constructed such that there is exactly one solution. An example of a Sudoku puzzle and its solution is given in Figure 3.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Fig. 1. A Sudoku puzzle and its solution [Wiki1]

In a classroom, one could discuss strategies for solving such a puzzle after presenting the problem. There are various methods, ranging from the identification of unique candidate digits for a certain field, via placement of digits with backtracking, to various heuristic and brute force methods. Here, one could discuss the complexities of various methods.

We assume that the students have a basic understanding of propositional logic and the notion of satisfiability. Probably it is a good idea to remind them that SAT is the generic NP-complete problem. With this harness, it is possible to describe the problem of finding a solution for a given Sudoku puzzle as a boolean satisfiabilty problem. The following encoding is inspired by the Sudoku web page and interactive Java applet by Ivor Spence [Spence].

We construct a boolean formula which is satisfiable if and only if the Sudoku puzzle has a solution. Assume that the rows and columns of the board are numbered from 1 to 9. In our formula, there are nine propositions per field: Proposition `p[`i`,`j`,`k`]` indicates that the field on row i and column j contains digit k. Thus, in the above example, `p[1,1,5]` is `TRUE`, and `p[1,1,1]` is `FALSE`. Since there are $9 \times 9 = 81$ fields, this gives 729 propositions in total.

Obviously, it is tedious to write boolean formulas with these many variables explicitly. Therefore, we use an array notation `p[i,j,k]` together with finite quantification `forall i=1..9` and `exists i=1..9` as abbreviations for the respective conjunction and disjunction, respectively. For example, `exists k=1..9 (p[1,1,k])` is short for `(p[1,1,1] | p[1,1,2] | p[1,1,3] | p[1,1,4] | p[1,1,5] | p[1,1,6] | p[1,1,7] | p[1,1,8] | p[1,1,9])`[3]. It is easy to write an appropriate pre-processor which expands formulas containing these finite quantifications into purely propositional formulas.

The following formula asserts that each field contains exactly one digit:

```
( forall i=1..9 forall j=1..9 exists k=1..9 (p[i,j,k]) &
  forall i=1..9 forall j=1..9 forall k=1..9 forall l=1..9, l≠k:
                                         !(p[i,j,k] & p[i,j,l]))
```

The following formula asserts that each digit is contained in each row at least once:

```
( forall i=1..9 forall k=1..9 exists j=1..9 (p[i,j,k]) )
```

Similarly, it can be formulated that each digit is contained in each column at least once:

```
( forall j=1..9 forall k=1..9 exists i=1..9 (p[i,j,k]) )
```

The formula for the sub-squares looks slightly more complex, but its boolean expansion is not:

```
( forall s=0..2 forall t=0..2 forall k=1..9
            exists i=1..3 exists j=1..3 (p[(3*s+i),(3*t+j),k]) )
```

Recall that the expansion of this formula is as follows:

```
(p[1,1,1] | p[1,2,1] | p[1,3,1] | p[2,1,1] | ... | p[3,3,1]) &
(p[1,1,2] | p[1,2,2] | p[1,3,2] | p[2,1,2] | ... | p[3,3,2]) &
  ... &
(p[1,1,9] | p[1,2,9] | p[1,3,9] | p[2,1,9] | ... | p[3,3,9]) &
(p[1,4,1] | p[1,5,1] | p[1,6,1] | p[2,4,1] | ... | p[3,6,1]) &
  ... &
(p[7,7,9] | p[7,8,9] | p[7,9,9] | p[8,7,9] | ... | p[9,9,9])
```

Even though this may look like a long formula, it contains only $9^3 = 729$ literals. Finally, the value of pre-filled fields can be easily expressed by fixing the values

[3] In this paper, we use the symbols `|`, `&` and `!` for logical disjunction, conjunction, and negation.

of the respective propositions. For the above example, this gives

```
p[1,1,5] & p[1,2,3] & p[1,5,7] & p[2,1,6] & ... & p[9,9,9].
```

Now, the Sudoku puzzle is solvable if and only if the conjunction of the above formulas is satisfiable, i.e., if there exists an assignment of truth values to propositional variables such that the whole formula becomes true. Modern SAT solver like miniSAT are able to find such an assignment, if it exists, within a few seconds. The above formula contains 729 propositions and approximately 9.000 literals. For such a problem size, satisfiability is decided within a few seconds. Each satisfying model gives a solution of the puzzle, since it determines the value of all propositions in the formula.

It is important to remark that the formula is just a representation of the problem; solving the problem is completely done within the SAT solver. We did not suggest or implement any heuristic or systematic way to solve the puzzle. Therefore, this approach is applicable for a large range of similar examples, where the task is to construct a certain state. However, if the problem size increases, it may be of advantage to explore different strategies implemented within the specific SAT solver which is used. For this, various extensions of the problem (an $n \times n$ board with increasing n, additional conditions on the solution, latin squares, etc.) can be analysed.

Thus, in a verification course, SAT procedures and heuristics for SAT solving can be discussed starting with this example. Furthermore, the behaviour of different SAT solvers on various examples can be measured and discussed.

4 Solitaire puzzles and model checking

A somewhat more intricate sort of puzzles is asking not for a single goal state, but for a sequence of states leading to a certain outcome.. In a solitaire game, a single player has to construct this sequence. The case when there is more than one player will be covered in the next section.

A classical example from the late 9^{th} century is the 'wolf, goat and cabbage' problem. It is originally described as follows (see [AoY]): "A certain man needed to take a wolf, a she-goat and a load of cabbage across a river. However, he could only find a boat which would carry two of these [at a time]. Thus, what rule did he employ so as to get all of them across unharmed?"

This puzzle is excellent to introduce the notion of a labelled transition system. If we assume that the man and the boat are always at the same side of the river, then there are four state variables indicating the position of the items: `p_man, p_wolf, p_goat, p_cabb`. Each of these state variables can be `FALSE` (indicating that the respective subject is still on the original side of the river), or `TRUE` (indication that the river has been crossed). Thus, there are $2^4 = 16$ states in the labelled transition system. These are depicted in Fig. 4, where each node label gives the values of `p_man, p_wolf, p_goat, p_cabb`. For example, node label `1100` means that man and wolf have crossed the river, whereas goat and cabbage are still on the original side. Labels `w, g, c,` and `n` indicate a crossing of the river by the ferryman with wolf, goat, cabbage, or no load, respectively.

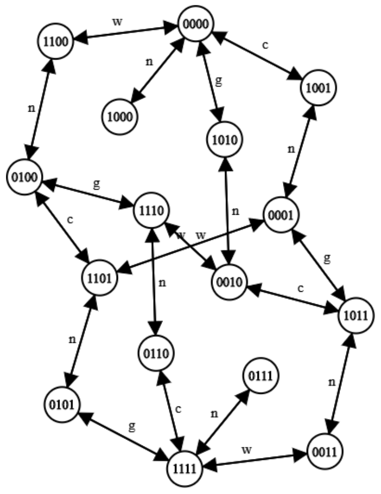


Fig. 2. Labelled transition system for the wolf-goat-cabbage problem

We want to model the problem in SMVL, the input language of the nuSMV model checker (see [CCGR 00]). As SMVL does not allow labels on transitions, we introduce an additional state variable `boat`, which can take any of the values {`w, g, c, n`}. With this, the transition relation is described as follows:

```
init(p_man) := FALSE;
init(p_wolf):= FALSE;
init(p_goat):= FALSE;
init(p_cabb):= FALSE;

next(p_man):= ! p_man ;
next(p_wolf):= case
  p_wolf = p_man & boat = w : !
p_wolf;
  TRUE : p_wolf; esac;
next(p_goat):= case
  p_goat = p_man & boat = g : !
p_goat;
  TRUE : p_goat;
esac;
next(p_cabb):= case
  p_cabb = p_man & boat = c : !
p_cabb;
  TRUE : p_cabb;
esac;
```

Here are some explanations on this definition. Initially, all state variables are `FALSE`, and the variable `boat` has an arbitrary value. The variable `p_man` oscillates between `TRUE` and `FALSE`; we assume, that the ferryman crosses the river in every step. The next position of the wolf is determined as follows: If the wolf is on the same side as the man, and the `boat` variable indicates that the wolf is to be taken, then in the next state the wolf will be on the other side. In all other cases, the wolf has to remain where it is, i.e., the variable does not change. The same explanation holds for the lines concerning goat and cabbage.

In order to solve the puzzle automatically, we need to express the fact that certain of the states are "harmful". This can be conveniently done by a boolean formula on the state variables:

```
  wolf_eats_goat := ((p_wolf = p_goat) & (p_man != p_wolf));
  goat_eats_cabb := ((p_goat = p_cabb) & (p_man != p_goat));
  harmful := (wolf_eats_goat | goat_eats_cabb);
```

This reflects the fact that the wolf can eat the goat, if they are both on the same side, and the ferryman is on the other side; and likewise for goat and cabbage. Furthermore, we can characterize the goal state(s):

```
  all_crossed := (p_man& p_wolf & p_goat & p_cabb);
```

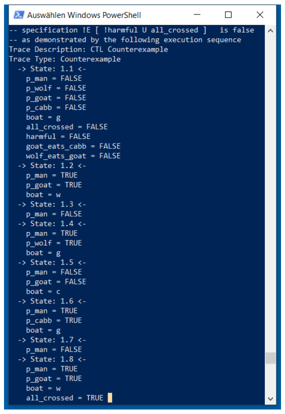
The task is completed if all three objects have crossed the river. Now we can ask nuSMV for the solution of the puzzle:

```
  SPEC E [!harmful U all_crossed]
```

`E` $[\varphi$ `U` $\psi]$ is the CTL *existential-until* operator. Thus, the formula can be read as "there is a path from an initial state to a state where all items have crossed which passes no harmful state." The model checker immediately confirms that this is the case. However, this is not very helpful, as we would like to know an example for such a path. Therefore, we ask for the negation of the formula:

```
  SPEC !E [!harmful U all_crossed]
```

As expected, the model checker confirms that this formula is false and delivers as proof a counterexample, i.e., a sequence of eight steps falsifying the formula and thus demonstrating our original aim.



```
-- specification !E [ !harmful U all_crossed ]   is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    p_man = FALSE
    p_wolf = FALSE
    p_goat = FALSE
    p_cabb = FALSE
    boat = g
    all_crossed = FALSE
    harmful = FALSE
    goat_eats_cabb = FALSE
    wolf_eats_goat = FALSE
  -> State: 1.2 <-
    p_man = TRUE
    p_goat = TRUE
    boat = w
  -> State: 1.3 <-
    p_man = FALSE
  -> State: 1.4 <-
    p_man = TRUE
    p_wolf = TRUE
    boat = g
  -> State: 1.5 <-
    p_man = FALSE
    p_goat = FALSE
    boat = c
  -> State: 1.6 <-
    p_man = TRUE
    p_cabb = TRUE
    boat = g
  -> State: 1.7 <-
    p_man = FALSE
  -> State: 1.8 <-
    p_man = TRUE
    p_goat = TRUE
    boat = w
    all_crossed = TRUE
```

Fig. 3. nuSMV solution for the wolf-goat-cabbage problem

The above example has the disadvantage that it is not easy to extend the problem size. Grid-based games and puzzles like Sudoku are better suited to explore the abilities and limits of verification tools. A well-known example of this kind, allegedly invented by Sam Loyd in the 1870s, is the Fifteen-puzzle, It consists of a $h \times v$ grid in which there are $(h \cdot v) - 1$ numbered tiles and one blank space. Originally, $h = v = 4$, hence the name of the puzzle. A move consists in moving any tile into the position of the blank. The goal is to achieve a certain predetermined order on the tiles (usually ascending).

In contrast to Sudoku, this puzzle can not directly be coded as a boolean satisfiability problem. The set of states is given by the distribution of the tiles on the grid. Similar as in the previous example, solving the puzzle must be done by constructing a sequence of states, where each next state is reachable from the current state via a legal move.

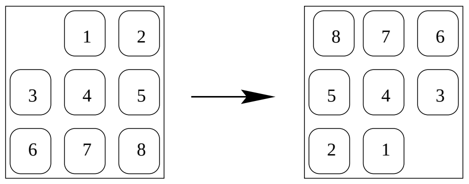


Fig. 4. The 3×3–Fifteen-puzzle: Start- and end-state

This puzzle can be described by a state-transition system as follows. For each tile there is a program variable which notes its horizontal and vertical position. Furthermore, there is a program variable `move` indicating whether the next move will be a shift up, down, left or right of the blank space. If the move would bring it out of the borders, nothing is changed; otherwise, its position is swapped with the respective adjacent tile.

The SMV code corresponding to this description[4] is shown below.

For $h = 3$ and $v = 3$, the internal representation of the transition relation uses 440.419 nodes. There are $4 \cdot (h \cdot v)! = 1.4 \cdot 10^6$ states, of which 50% are reachable from an initial state. (Note that there are four initial states, since we did not fix the initial value for variable `move`.) As in the previous example, the specification claims that a certain final state is *not* reachable; the model checker contradicts this claim by showing a sequence of moves (ddrruullddrruullddrruullddrr) which gives a solution to the puzzle. The solution is found within a few seconds.

[4] As above, in the actual SMV code, variable array bounds or indices, e.g., vpos[i], are not allowed and have to be replaced by the respective constant values vpos[1],vpos[2],...

```
MODULE main
 DEFINE h := 3; v := 3;
 VAR move: {u,d,l,r};
     hpos: array 0..(h*v-1) of 1..h;
     vpos: array 0..(h*v-1) of 1..v;
 ASSIGN
 next(vpos[0]) := case
   (move=l) & !(vpos[0]=1) : vpos[0] - 1;
   (move=r) & !(vpos[0]=v) : vpos[0] + 1;
   1: vpos[0]; esac;
 next(hpos[0]) := case
   (move=u) & !(hpos[0]=1) : hpos[0] - 1;
   (move=d) & !(hpos[0]=h) : hpos[0] + 1;
   TRUE: hpos[0]; esac;
 forall i=0..8:
 next(vpos[i] := case
   (move=l) & !(vpos[0]=1) & hpos[i]=hpos[0] & vpos[i]=vpos[0]-1 |
   (move=r) & !(vpos[0]=v) & hpos[i]=hpos[0] &
vpos[i]=vpos[0]+1 : vpos[0];
   TRUE: vpos[i]; esac;
 next(hpos[i]) := case
   (move=u) & !(hpos[0]=1) & vpos[i]=vpos[0] & hpos[i]=hpos[0]-1 |
   (move=d) & !(hpos[0]=h) & vpos[i]=vpos[0] &
hpos[i]=hpos[0]+1 : hpos[0];
   TRUE: hpos[i]; esac;
 init(vpos[i]) := i div v; init(hpos[i]) := i mod v;
 DEFINE goal := (vpos[i] = 3 - (i div v) & hpos[i] = 3 - (i mod v));
 SPEC !EF goal
```

For $h = 4$, $v = 3$, there are approximately 10^9 reachable states. Although the symbolic model checker detects rather quickly that some solution must exist, for the construction of a concrete solution sequence the state space has to be partitioned into strongly connected components. This requires significant CPU time and memory. Thus, this example is well-suited to discuss complexity and try the many options which nuSMV offers. In particular, a teacher can explain the idea of bounded model checking, thereby relating solitaire games to SAT solving.

There are several other, comparable puzzles of this type which can be treated in classroom exercises. An example similar to river crossing is the 'water pouring puzzle' from the 17th century, where the objective is to reach a certain distribution of water in three jugs [Bac 1612]. Various sliding block puzzles like Klotski or Sokoban can be treated similar to the 'Fifteen-puzzle'. Another example is peg solitaire, where the objective is to empty a board of pegs [JMMT 06].

5 Board games and strategic logics

In the previous section, we considered puzzles and solitaire games, which are well-suited of demonstrating the computing paradigm of *reactive systems*. A single player reacts to a challenge posed by the environment. The situation changes if we consider *interactive systems*, where two or more players interact, and the actions of the players are mutually dependent. Interactive systems have been researched under different names: Distributed computing systems, Multi-agent systems, Cyber-physical systems, Multi-player games, and others.

For modelling interactive systems, *concurrent game structures* (CGS) are being used. Basically, a concurrent game structure for n players is an n-tuple of labelled transition systems. Formally, a CGS is a structure $(Agt, S, Act, \pi, \delta, s_0)$, where $Agt = \{a_1, ..., a_n\}$ is a finite set of *agents*, $S = S_1 \times \cdots \times S_n$ is a finite set of *global states* (and each global state s is a tuple $\langle s_1, ..., s_n \rangle$ of local states), Act is a finite set of *actions*, $\pi : Agt \times S \mapsto 2^{Act}$ is the *protocol function* indicating which actions are available to an agent is a certain state, $\delta : S \times Act^n \times S$ is the *evolution function* (i.e., transition relation) indicating how the global state changes if the agents each perform a certain action, and $s_0 \subseteq S$ are the initial states of the structure. The evolution function gives a successor state only for those combination of actions which are available to the agents in a state (i.e., $(s, \langle e_1, ..., e_n \rangle, s') \in \delta$ implies that $e_i \in \pi(a_i, s)$ for all $i \leq n$, and if $e_i \in \pi(a_i, s)$ for all $i \leq n$, then there is exactly one s' such that $(s, \langle e_1, ..., e_n \rangle, s') \in \delta$).

A *strategy* is a plan which tells an agent which of the available actions to choose in a certain situation. Often, the strategy is used to reach a designated goal state, or to stay within a set of safe states. Formally, a strategy σ_i for agent $a_i \in Agt$ is a function $\sigma_i : S \mapsto Act$, such that $\sigma_i(s) \in \pi(sa_i, s)$. Given strategies $\sigma_1, ..., \sigma_n$ for all players of a CGS, there is exactly one execution sequence following all these strategies.

Strategic logics like the alternating temporal logic ATL have been proposed to reason about interactive systems modelled by concurrent game structures. ATL is an extension of CTL which allows quantification on strategies. Formula $\langle A \rangle \varphi$ expresses that there exists a strategy for the players $a \in A$ such that they can force the temporal logic formula φ to hold, no matter what the other players do, This is convenient to express the fact that a player has a winning strategy in a game.

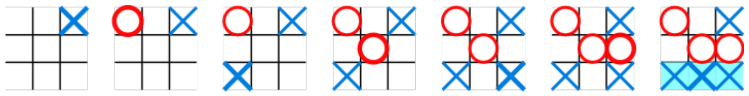


Fig. 5. A Tic-tac-toe game [Wiki2]

As a simple example, we use the classic game of Tic-tac-toe, also called 'Noughts and crosses' or 'Three in a row'. It is played on a 3×3 grid on paper, where two players take turn marking the fields with X and O. The player who first has three of her/his symbols horizontally, vertically or diagonally in a row wins. For an example game, consider Fig. 5.

In contrast to solitaire games, in two-player games the actions of each player depend on the actions of the opponent. Thus, to construct a winning strategy one has to consider all potential strategies of the opponent. This makes strategic model checking often much more complex than reachability analysis.

Such games can be analyzed via strategic model checkers such as MCMAS (see [LQR 09]). Subsequently, we formalize the rules of Tic-tac-toe in (a language similar to)[5] ISPL, which is the input language for MCMAS.

```
Agent Environment
  Obsvars:
    turn: {nought, cross}; -- variable indicating who's turn it is
    b[1..3][1..3]: {x, o, b}; -- the board markings; b means blank
  end Obsvars
  Evolution:
    -- turn switches between every two moves
    turn=nought if turn=cross; turn=cross if turn=nought;
    -- board is marked according to the move
    forall i=1..3 forall j=1..3
      b[i][j] = o if turn = nought & Nought.Action = a_ij;
    forall i=1..3 forall j=1..3
      b[i][j] = x if turn = cross  & Cross.Action  = a_ij;
  end Evolution
end Agent

Agents Nought, Cross
  Actions = {a_ij | i=1..3, j = 1..3};
  Protocol: -- Action a_ij is available if Board b[i][j] is blank:
    forall i=1..3 forall j=1..3 ( b[i][j]=b : {a_ij} );
  end Protocol
end Agent

Evaluation
  noughtwins if
    ( exists i=1..3 b[i][1]=o & b[i][2]=o & b[i][3]=o ) |
    ( exists i=1..3 b[1][i]=o & b[2][i]=o & b[3][i]=o ) |
    b[1][1]=o & b[2][2]=o & b[3][3]=o |
    b[3][1]=o & b[2][2]=o & b[1][3]=o;

  crosswins if
    -- similar, =x instead of =o
end Evaluation

InitStates
  ( forall i=1..m forall j=1..n b[i][j]=b )  & (turn = cross);
end InitStates
```

[5] ISPL does not admit arrays, these must be expanded by a suitable pre-processor. Furthermore, ISPL has some syntactic peculiarities which do not contribute to the goals of this article and, thus, are left out. The full code of all examples in this article can be obtained from the author.

```
Formulae
  <Cross> F crosswins;
  <Nought> F noughtwins;
end Formulae
```

Perhaps surprisingly, the model checker reports that the first formula is `TRUE` and the second one `FALSE`. This is because agent `Cross` indeed has a strategy to reach three `x` in a row if it does not care whether agent `Nought` reaches three `o` in a row first. Thus we have to ask whether one of the following formulas is true:

```
<cross> F (crosswins & ! noughtwins);
<nought> F (noughtwins & ! crosswins);
```

This is indeed not the case, as the model checker confirms in 0.372 s.

Tic-tac-toe can be easily generalized to the (m, n, k)-*game*, where two players compete to place k symbols in a row on an $m \times n$ grid. This generalization is mathematically interesting, as it quickly leads to open questions: For example, for $k \geq 9$ it can be shown that even on an infinite board there is no winning strategy for the first player. However, for $k = 6$ or $k = 7$ it is not known whether there are m and n such the first player has a winning strategy in the (m, n, k)-game. Consequently, the complexity of model checking quickly grows with increasing m, n and k: Already for the $(4, 4, 3)$-game there are more than 10^7 reachable states, and it takes more than 4 minutes to find a winning strategy for agent `Cross`.

Thus, for use in a model checking course, we prefer other examples. Subsequently, we describe a location-based game which has been called Tourality[6]. It is played by two players and resembles the classical computer game PacMan, but without the real-time aspect.

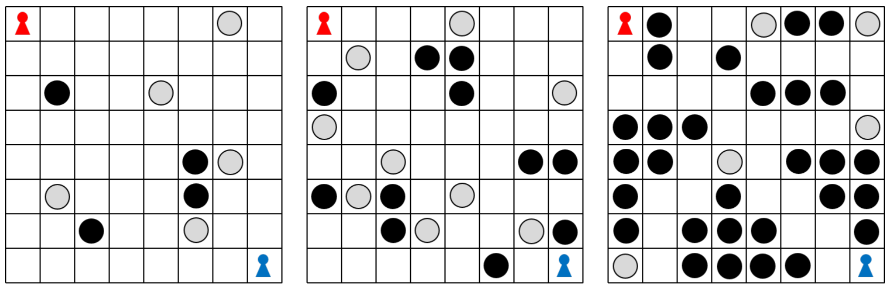


Fig. 6. Three different Tourality set-ups

[6] Actually, Tourality is a GPS-based treasure hunt where people are trying to outperform each other in reaching certain locations. We use a card-board abstraction of this game which was introduced in a German national competition for computer science education.

The game is played on an 8×8 droughts / checkers board, with some set-up of black and white tiles. Black tiles are obstacles, which remain in position throughout the game. White tiles are rewards, which are collected by the players. In one corner of the board there is a red peg, and in the opposite corner a blue peg. One player sets up the board, the other one chooses whether to play red or blue. Players take turn moving their peg to an (horizontally or vertically) adjacent field on the board. It is not allowed to move onto a field occupied by an obstacle or the other player's peg. If a player moves the peg onto a field where there is a reward, it is collected. Red begins. The game ends when all rewards are collected, and the player who has collected most rewards wins.

Tourality can be easily modelled by a concurrent game structure. For space reasons we give only the main constituents, again in "not quite" ISPL.

```
Agent Environment
  Obsvars:
    turn : {red, blu}; xred, yred, xblu, yblu : 1..8;
    reward[1..5] : {avail, taken}; -- for example, 5 rewards
    points_red, points_blu: 0..5;
    constant b[1..8][1..8] : {empty, block}; -- the board
    constant xreward[1..5], yreward[5]: [1..5]; -- positions of the rewards
  end Obsvars
  Evolution:
    -- turn switches between every two moves
    turn=red if turn=blu; turn=blu if turn=red;
    -- positions are updated according to the move
    yred=yred-1 if turn = red & Red.Action=up;
    yred=yred+1 if turn = red & Red.Action=dn;
    -- and similar for other actions and player Blu;
    -- board and points are updated according to the moves of the players:
    for 1=1..5: reward[i] = taken if reward[i] = avail &
      (turn = blu & xred = xreward[i] & yred = yreward[i] |
       turn = red & xblu = xreward[i] & yblu = yreward[i]);
    for 1=1..5: points_red=points_red+1 if reward[i] = avail &
      turn = blu & xred = xreward[i] & yred = yreward[i];
    for 1=1..5: points_blu=points_blu+1 if reward[i] = avail &
      turn = red & xblu = xreward[i] & yblu = yreward[i];
  end Evolution
end Agent

Agent Red
  Actions = { up, dn, lt, rt };
  Protocol:
    -- if it is red's turn and at position i,j and target field is not blocked
    -- and target field is not occupied, then the movement action is available
    for some x=1..8: for some y=1..7:
      xred=x & yred=y & b[x,y+1]=empty & !(xblu=x & yblu=y) : { dn };
    for some x=1..8: for some y=2..8:
      xred=x & yred=y & b[x,y-1]=empty & !(xblu=x & yblu=y) : { up };
    for some x=1..7: for some y=1..8:
      xred=x & yred=y & b[x+1,y]=empty & !(xblu=x & yblu=y) : { rt };
    for some x=2..8: for some y=1..8:
      xred=x & yred=y & b[x-1,y]=empty & !(xblu=x & yblu=y) : { rt };
   end Protocol
end Agent

Agent Blu
  -- similar
end Agent

InitStates
  b = [[empty, empty, block, empty, block, empty, empty, empty], ...] &
  xreward=[3,2,5,4,3], yreward=[2,5,7,1,4] &
  xred = 1 & yred = 1 & xblu = 8 & yblu = 8 & turn = red &
  for i=1..5: reward[i] = avail & points_red = 0 & points_blu = 0;
end InitStates

Formulae
  <Red> F (points_red >= 3); -- has Red a winning strategy?
end Formulae
```

MCMAS can check the example set-ups in Fig. 5 within a few seconds (for the first and third board) and a few minutes (for the second one). As can be seen by these examples, already the normal rules of the game allow for many variants. One can arrange the obstacles such that they form a maze, or such that they block certain parts of the board. This can drastically reduce the number of reachable states. Alternatively, one can have few or many rewards on the board, thereby decreasing or increasing the number of possible strategies. Already a few experiments show this effect very clearly: whereas the first example setup in Fig. 5 has 7×10^5 reachable states and uses $3.5s$, the third one has only 3×10^5 states, but needs $12s$. Adding more rewards yields an exponential blowup, e.g., the middle setup in Fig. 5 with 9 rewards has 2×10^7 reachable states and uses $6m$, with 10 rewards the size of the reachable state space is 10^8 and uses $15m$. and with 11 rewards there are 10^9 reachable states calculated in $150m$.

However, in a computer based version of the game, it is also possible to vary other parameters of the game. For example, the vicinity relation can be changed to allow also horizontal moves. Alternatively, one can allow only moves according to the knight's movement in chess. Furthermore, one can replace the turn-based move by concurrent moves of the player. It is also possible to introduce more than two players, and to allow coalitions and competition between them. It is fun to play around with these alternatives and see how the game changes and how the model checker behaves. This can be useful to explain the potential of strategic model checking.

Other locality-based games are likewise well-suited to motivate this technique. For example, formalising the rules of Nine Men's Morris is an interesting exercise. A more challenging task would be to ask students to model and solve end games in chess.

6 Conclusion

In this paper, we have shown how to use puzzles and games to demonstrate the potential and limitations of model checking in a graduate-level course. We used SAT solving for Sudoku, model checking for the 'wolf, goat and cabbage' problem and the Fifteen-puzzle, strategic model checking for Tic-tac-toe and Tourality, and suggested to use card games for the introduction of epistemic logics.

There are many extensions to the ideas presented here. SMT solving (satisfiability modulo theories) can be used for problems involving integers. An example are alphametic puzzles, where an arithmetic problem is given with letters in place of the digits, and the challenge is to deduce which digit corresponds to each letter.

Discrete Markov chains and probabilistic model checking (e.g., with the PRISM model checker) can be explained with the help of dice games.

An extension to model checking of multi-player games with perfect information is the case of imperfect information. To model such situations, epistemic concurrent game structures have been proposed. MCMAS is able to encode individual knowledge of the agents in the epistemic accessibility relation between states. Furthermore, formulas containing epistemic modalities **E**, **C**, and **D** can be checked. Epistemic logics are often introduced via artificial set-ups such as dining cryptographers ("Who knows what about the payment?"), muddy children ("Who has a spot on the forehead?"), detective puzzles ("Does the murderer know that the detective knows that ..."), or similar. We prefer to use simplified versions of common card games like Bridge, Poker, or Blackjack, where each player has some private knowledge, and additionally there is public knowledge about the distribution of cards. However, a strong factor in these games is the probabilistic aspect brought in by the random distribution of cards. Unfortunately, to our knowledge there are no logics and tools combining probabilistic analysis with strategic and epistemic reasoning. This could turn out to be a fruitful research topic.

Thus, our considerations are not only useful for education. Since games and puzzles often are just abstract representations of real-life challenges, they may give directions on how to further evolve the formal methods tools. As another example, we would like to be able to use strategic model checking in an on-line fashion, to enable the synthesis of controllers for embedded systems which have to react in real time.

Acknowledgements: The author wishes to thank Damian Kurpiewski for helpful discussions on MCMAS, and Stephan Merz for discussions on the game-based approach to model checking education. This research was supported by the German BMBF project "CrESt" on collaborative embedded systems, FKZ 01|S16043G/E.

References

[BrMus] British Museum: The Royal Game of Ur ~2600 / ~2400. https://artsandculture.google.com/asset/the-royal-game-of-ur/MwE2MMZNSKiTwQ
Clay cuneiform tablet. https://www.britishmuseum.org/research/collection_online/collection_object_details.aspx?objectId=796973&partId=1

[Tur 50] Alan Turing: Computing Machinery and Intelligence. Mind **59**, 433–460, Oxford University Press (1950).

[SHS+ 17] David Silver et al.: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815v1 [cs.AI] (5 Dec 2017), https://arxiv.org/pdf/1712.01815.pdf

[SY01] N. V. Shilov and K. Yi: Puzzles for Learning Model Checking, Model Checking for Programming Puzzles, Puzzles for Testing Model Checkers. Electronic Notes in Theoretical Computer Science 43 (2001) http://www.elsevier.nl/locate/entcs/volume43.html

[vDR07] H. van Ditmarsch and J. Ruan: Model Checking Logic Puzzles. 2007 https://www.researchgate.net/publication/29646125_Model_Checking_Logic_Puzzles

[EJV05] J. Eskildsen, L.H. Jensen and B.M. Vester: Symbolic Model Checking in Puzzle Games – Automated Reachability Analysis. Aalborg Universitet, DAT4, May 2005. https://pdfs.semanticscholar.org/1783/d58420fd6a05b90dc29bbba5fc2cd9e3a113.pdf

[Wiki1] Wikipedia: Sudoku, https://en.wikipedia.org/wiki/Sudoku. Puzzle drawn by Tim Stellmach, CC0, https://commons.wikimedia.org/w/index.php?curid=57831926, solution drawn by en:User:Cburnett, CC BY-SA 3, https://commons.wikimedia.org/w/index.php?curid=57831971

[Spence] Ivor Spence: The SuDoku Puzzle as a Satisfiability Problem. http://www.cs.qub.ac.uk/~I.Spence/SuDoku/SuDoku.html

[AoY] Alcuin of York: Propositiones ad Acuendos Juvenes, Problem XVIII. "Propositio de homine et capra et lupo." 9th century a.D., Translation by P. Burkholder, http://www.math.muni.cz/~sisma/alcuin/anglicky1.pdf

[CCGR 00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri: NuSMV: A New Symbolic Model Verifier. Int. J. STTT **2**.4, pp. 410–425 (2000)

[Bac 1612] Claude-Gaspard Bachet: Problemes plaisans et delectables, Published by P. Rigaud, Lyon (1612). Text available at https://www.loc.gov/resource/rbc0001.2009gen48833/. See also https://en.wikipedia.org/wiki/Water_pouring_puzzle

[JMMT 06] C. Jefferson, A. Miguel, I. Miguel, S.A. Tarim: Modelling and solving English Peg Solitaire. Computers & Operations Research **33**.10, pp. 2935-2959 (2006)

[CS01] E.M. Clarke, H. Schlingloff: Model Checking. In: Handbook of Automated Reasoning, Elsevier Science Publishers (2001)

[Wiki2] Wikipedia: Tic-tac-toe. https://en.wikipedia.org/wiki/Tic-tac-toe. Drawing by User:Stannered - en:Image:Tic-tac-toe-game-1.png, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1866155. See also https://en.wikipedia.org/wiki/M,n,k-game

[LQR 09] A. Lomuscio, H. Qu, F. Raimondi: MCMAS – A Model Checker for the Verification of Multi-Agent Systems. Int. Conf. Computer Aided Verification, pp 682-688 (2009)
All URL accessed Nov 13th, 2019