IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 30 May 2022, accepted 17 June 2022, date of publication 21 June 2022, date of current version 27 June 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3185109

Verifying Maze-Like Game Levels With Model Checker SPIN

ONUR TEKIK[1], ELIF SURER[2], (Member, IEEE), AND AYSU BETIN CAN[1]
[1]Department of Information Systems, Graduate School of Informatics, Middle East Technical University, Ankara 06800, Türkiye
[2]Department of Multimedia Informatics, Graduate School of Informatics, Middle East Technical University, Ankara 06800, Turkey

Corresponding author: Aysu Betin Can (betincan@metu.edu.tr)

ABSTRACT This study presents a framework that procedurally generates maze-like levels and leverages an automated verification technique called model checking to verify and produce a winning action sequence for that level. By leveraging the counterexample generation feature of the SPIN model checker, one or more solutions to the level-in-test are found, and the solutions are animated using a video game description language, PyVGDL. The framework contains four behavioral templates developed to model the logic of maze-like puzzle games in the modeling language of SPIN. These models automatically are tailored according to the level-in-test. To show the proposed methodology's effectiveness, we conducted five different experiments. These experiments include performance comparisons in level-solving between the proposed and existing methodologies —A* Search and Monte Carlo Tree Search— and demonstrations of the use of the proposed approach to check a game level with respect to requirements. This study also proposes a pipeline to generate maze-like puzzle levels with two levels of cellular automata.

INDEX TERMS Formal verification, model checking, procedural content generation, puzzle games, video game description language.

I. INTRODUCTION

Procedural generation of video game levels, which is the automated process of algorithmically creating game levels, has been an essential topic in game research [1]. Part of this importance comes from the ultimate goal of having unlimited content for games. However, having unlimited content has its downsides, such as the resources needed to verify whether a level fits a specific requirement. As in the generation of the game levels, verification of the game levels also requires computational resources.

A common practice for verifying whether a game level meets specific requirements is called ''Playtesting,'' which may include intense human labor or computational time since this process is not automated. It can be semi-automated [2], or it can be done via memory and CPU-intensive methods [3], [4]. This paper proposes another method for verifying maze-like game levels —an automated and model-based approach.

Another practice used to verify a game is having automated tests. Although automated tests can speed up the process significantly, test automation is still labor-intensive in designing and coding test scenarios. Model-checking is a push-button technique similar to automated tests. A comparative study showed that model checking is compatible, even outperforms, the bug-finding capabilities of testing tools [5]. Using a model checker requires creative effort in modeling the behavior of a system similar to the creative effort required in designing test scenarios. This paper presents behavioral models designed for puzzle games to reduce such effort.

A puzzle game is a system that has a set of game rules that can be rendered into a behavioral model, an initial configuration, which in this case is a game level, and game-ending or game-breaking conditions that can be specified in a linear temporal logic (LTL) formula. All these characteristics make it possible to model and formally verify puzzle games with the help of a model checker [6]. One advantage of model checking is automated counterexample generation when a sequence of actions violates a specified requirement. We leverage this feature to generate game-winning action sequence by instructing the model checker to verify that the winning state is unreachable.

We focus on three puzzle games: maze for single player, racing in a maze against a non-player character (NPC), and Sokoban game, in which the player must push several boxes

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

into a set of designated storage locations without getting itself or the boxes stuck [7]. The behavioral logic of these games is modeled PROMELA, the input language of the SPIN model checker, as our first and foremost contribution. There is one model for each game except Sokoban. There are two models for Sokoban, one that considers every sprite position change and one that considers only the changes in positions of boxes. Given a game level, our framework tailors the model based on the avatar's initial map structure and position and then checks game requirements, including that the game is winnable. The counterexample generated by the model checker is not human readable. The framework translates this trail of execution into automatically playable instructions on the game environment as our next contribution.

The games in this paper are from the General Video Game Artificial Intelligence (GVG-AI) framework [8], which contains hundreds of two-dimensional games that are defined in Video Game Description Language (VGDL) [9]. The GVG-AI framework and VGDL are frequently used in computational intelligence and game research. Using cellular automata, our framework generates maze and race game levels defined in PyVGDL. The Sokoban game's levels were exported from the website by Garcia [10]. Then, the behavioral model templates are automatically tailored according to the game level and checked using SPIN. The sequence generated by the model checker is translated into actions playable on GVG-AI.

This study's contributions can be summarized as follows: This research presents an end-to-end pipeline to verify and create maze-like puzzle games. The algorithmic contribution focuses on modeling the game logic in detail —a crucial abstraction— and simplifying the formulae automatically based on this abstraction. The other contribution is the application of SPIN to a new domain, gaming, and the conversion of SPIN's outputs into playable results. Besides, this pipeline allows well-known algorithms and tools like SPIN to be rapidly integrated into the proposed pipeline while enabling the visualization of the results using tools like GVG-AI. This visualization aspect is also essential because SPIN is useful in software engineering; it is not widely used outside critical systems because SPIN does not provide visual outputs other than X-SPIN. The SPIN output is re-animated as playable levels on GVG-AI, allowing game developers to perceive the gameplay readily, and the solution is re-animated with visual cues. Finally, we give a well-structured, push-button solution based on game design-based attributes that adhere to specific rules.

The rest of the paper is organized as follows. Section 2 presents preliminaries on cellular automata, model checker SPIN, and the GVG-AI framework. Section 3 presents the literature review. Section 4 presents our methodology explaining the proposed pipeline starting from procedural level generation and sprite placement, followed by proposed behavioral models and verification of game-specific properties. Section 5 presents the experiments and their results. Finally, Section 6 concludes the paper.

II. PRELIMINARIES

This section introduces preliminary material on cellular automata, the model checker SPIN, PROMELA, and GVG-AI, that are used and referenced in this study.

A. CELLULAR AUTOMATA

A cellular automaton consists of a grid of cells in a shape that evolves through a number of steps. Each cell is in one of a finite number of states. A cell's next state is calculated with a fixed rule, called a "state transition function," which includes the states of the cell itself and its neighbors. The state transition function is applied to all cells simultaneously.

Types of cellular automata used in this study are elementary cellular automata [11] and a life-like cellular automaton that can create maze-like levels from a random seed called "Maze" [12]. Elementary cellular automata are the most straightforward kind of one-dimensional cellular automata. The states of the cells are binary, zero and one. The cells are put in a line, and the next state of any cell depends on its two neighbor cells' state and its own. Since three binary inputs affect a binary output, there are limited variations of elementary cellular automata, $2^{(2\times 2\times 2)} = 256$ in total. Those rules are grouped into three different categories regarding the output pattern. These categories are simple, oscillating, and chaotic. Simple rules result in a stable end state, oscillating rules result in an ending pattern with few states that repeat stably, and chaotic rules result in a pattern that is partly stable or oscillating but unpredictable at the same time.

"Maze" is a life-like automaton that can be used to generate a maze-like output [12]. Maze-like means that its output includes long and connected corridors that can be interpreted as a maze. The automaton works in the Moore neighborhood, including four main neighbors and four secondary neighbors (North, northeast, east, southeast, south, southwest, west, and northwest).

B. MODEL CHECKING, LINEAR TEMPORAL LOGIC, SPIN, AND PROMELA

Model-checking [13] is a computational technique used to automatically verify concurrent systems that either are finite-state or have finite-state abstractions. A behavioral model of the system and its requirements are given to a model checker as input. The model checker performs an exhaustive search on the state-space of the submitted model to find a sequence of execution that violates the requirement.

Requirements are expressed as temporal properties. One popular way is using linear temporal logic (LTL) [14]. Linear temporal logic formulae can be used to encode facts in time, such as a condition that will never or eventually be true or a condition that can never be true unless some other is. An LTL formula consists of propositional variables, logic, and temporal operators. Table 1 presents the most common temporal operators.

In this paper, the model checker is SPIN [15], which employs an automaton-theoretic model checking algorithm.

TABLE 1. Temporal operators in linear temporal logic.

Operator	Notation	Explanation
Next	**X** p	Property p must hold at the next state of the execution
Future	**F** p <> p	Property p must hold in the future of the execution eventually
Globally	**G** p [] p	Property p must hold at all times
Until	p **U** q	Property p holds until q becomes true. In the future q must be true
Weak Until	p **W** q	Property p holds until q becomes true. Property q may or may not hold in the future

The automata used in the algorithm is called Büchi automata which accept infinite words. In the model checking domain, an infinite word is an infinite execution sequence generated by the given system model. The algorithm works with two automata. One automaton represents the model of the system, while the other represents the LTL formula specifying a requirement the system must satisfy. Vardi and Wolper showed that any LTL formula could be translated into a Büchi automaton [16] which recognizes all sequences that satisfy the LTL formula.

The automaton-theoretic model checking algorithm is as follows. Let M be the Büchi automaton recognizing all of the execution sequences generated by the given system model. Let also F be the Büchi automata accepting the complement of the given LTL property we want the system to satisfy, i.e., the requirement. Let $L(B)$ represent the language of the automata B. The system does not violate the given property if $L(M) \cap L(F) = \varnothing$, the empty language. If the result is non-empty, then it is the execution sequence that is generated by the system and violates the property. The result of the intersection is a counterexample to be emitted by the model checker. If the intersection is an empty language, then the system model satisfies the property. The model checker SPIN employs this algorithm in a space and time-efficient manner [15].

SPIN, an open-source model checker tool, is a generic verification system that supports the design and verification of systems that include asynchronous processes [15]. SPIN has been used to verify mission-critical software, including medical device protocols and the automotive industry [17], [18]. SPIN accepts correctness properties expressed in linear temporal logic (LTL) and performs on-the-fly verification based on the automata-theoretic foundation explained above [19].

The input language of SPIN is called PROMELA (Process or Protocol Meta Language) [19]. A system behavior is modeled using this language. PROMELA is built to model processes in parallel systems. It realizes Dijkstra's Guarded Command Language. PROMELA code is written in a way where it can have more than one possible flow in a conditionally controlled structure. If more than one guarding condition is true at any point, this implies a non-deterministic situation. If SPIN is instructed to make a single simulation from the model, one of the possible flows is chosen with an equally random chance. However, if verification is in progress, as in our case, SPIN looks for every possible flow at every choice —not a random choice—with the help of its automata-theoretic model-checking algorithm. The language also includes a communication structure between processes called channels. A channel may either be a buffer-based or a rendezvous-based channel with zero buffering ability. A message sent to a rendezvous-based channel blocks the execution of a process until another process receives the message. This feature is used to model the synchronous communication among processes. Another feature is that it is possible to include C code snippets and C program files to PROMELA code for additional capabilities. These C code segments are accessible from the PROMELA side, and they can access any variable or process on the PROMELA side.

SPIN has directives for enabling some optimizations. These optimizations increase speed or reduce memory usage by using various techniques. Some examples of these directives are used in our work. The first one is using bitstate hashing, reducing state-space table size, and increasing memory effectiveness while searching. The second one is disabling array boundary violation checks, increasing the verification speed of SPIN. The third one is to generate more than one unique counterexample if possible. The last is a run-time option instructing SPIN to return a counterexample with the smallest number of state changes to generate the shortest solutions.

C. GVG-AI AND PyVGDL

GVG-AI is a framework that provides a common open-source platform for artificial intelligence and game researchers with hundreds of single and multi-player 2-D games [8]. To have a standard notation, a common Video Game Description Language (VGDL) was proposed in 2013 [9], called PyVGDL. Nowadays, the project has moved to PyVGDL 2.0 [20]. Two different files define games in PyVGDL: one is a game rules file, including sprites included in the game, how they are represented in the game file, how they interact with each other, and when the game terminates. The other is a level file, a 2-D array representing sprites' positions when the game starts.

In our study, we have used PyVGDL to re-animate the solutions found by our methodology. We have forked the PyVGDL project on GitHub [20] to implement a predictable non-player character (NPC) type for the games that need a competitive and modelable opponent, such as the race game. The NPC type follows the A* algorithm to chase a target and a controller that takes all the moves it will do up-front. There are two reasons for the implementation of that specific NPC. The first is to make it predictable, making the same moves

in the game model and engine. We gain consistency between the actual game and its model by making it predictable. The second reason is to benchmark the model-based solution against a perfect opponent. By making its opponent play optimally in a two-player game, we can benchmark the optimality of the solution that the model-based approach gathers.

III. LITERATURE REVIEW

Model-checking is a technique used for formally verifying reactive and concurrent systems [13], [22]. The model checking process includes modeling a system's behavior and then fully exploring this behavior with respect to temporal logic formulae, where the formulae stand for system requirements. Although solutions using the model-checking approach tend to suffer from the exponential nature of the model-checking problem [13], model checking is used to study games, including major video game titles [23]–[26], simple single-player [27], [28], or multi-player games [29].

Although verification of software, in general, is an active area for research, verification of games is only partially investigated. Research in games regarding formal verification focuses on three main areas. The first of these areas is verifying the game itself by verifying game rules. Verification of a multi-player mobile game, Penguin Clash, is done with a model checker by Rezin *et al.* [29]. The work is focused on verifying the robustness and correctness of game rules. The game is verified against four different basic formulae that verify the game's requirements using a model checker. These requirements include both players' basic collision behavior and the winnability of the game. Also, the work proposes a pipeline to create verified games by placing a model-checking phase between game design and game integration. However, the study is built for verifying the game rules instead of content related to the game. Verification of an RPG game specified in a domain-specific language is done by Barroca *et al.* [30]. The game development is done in a model-driven way. The developer creates a model for a role-playing game as the starting point. Next, the game model is transformed into a model to be verified in an algebraic Petri net. After verification, the developer's model is transformed into code for a mobile game. Although the model is promising, the necessary steps to construct a successful game model are not explicitly stated. The game of Bomberman is verified using a discrete-event simulation in a model checking environment in another study [27].

Another area is automated game balancing. A multi-player game where a player picks a strategy against others is expected to balance strategies to have more player excitement. If one way of playing the game is strictly better than others, the other strategies become obsolete, and the game can lose its variety. Kavangh *et al.* studied automated game balancing with the help of a probabilistic model checker [28]. With the help of a model checker, their solution mimics how a player adapts between different strategies in a chain to see if any strategy is superior to others. Milazzo *et al.* studied extracting successful strategies and balancing game core mechanics with the help of statistical and probabilistic model checkers [31]. The work includes three case studies that answer a game design question via model-checking.

The final area of focus is verifying a storyline in a story-driven game. Holloway [26] focuses on pointing out logically incorrect parts in a game's storyline before the actual development of the game starts. Verifying educational adventure video games with the help of model checking is achieved by Moreno-Ger *et al.* [32]. The work focuses on scene-based educational adventure games. After the verification step, if the game fails in the model checker, an animation of how the game failed is generated from the counterexample trace from the model checker. However, our study differs from this one in one main aspect. Since the study of Moreno-Ger *et al.* focused on adventure games built on a scenario, there are no fixed game rules like our target genre. So, in their study, every different scenario must be verified with a different model. On the contrary, our study is focused on puzzle games, which have a fixed rule set and a level configuration. Thus, unlike the study of Moreno-Ger *et al.*, our target genre allows us to generate models from template models.

We used cellular automata to generate maze-like game levels in our main pipeline. Usage of cellular automata on procedural generation of game levels is also one of the popular research areas over the last decade [33]–[35]. The main reasons for the usage of cellular automata are their simplicity and reliability [33]. A simple cellular automata-based algorithm is evaluated for its performance in generating tunnel-based maps by Johnson *et al.* [33]. Cellular automata are also used with the help of genetic algorithms to generate mazes [34], [35]. There is also a pure cellular automata solution proposed [12]. Another methodology proposed to generate game levels includes multi-layered cellular automata and Hilbert curves [36]. There is also a different usage of cellular automata called self-referencing cellular automata [37]. The self-referencing cellular automata are developed to compensate for the lack of feedback between the elementary cellular automata rules and their output. The rule depends on the output line-by-line. The output and the rule of cellular automata may oscillate mutually, get fixed in a mutual point, or the output may be oscillating with a fixed rule. However, the output of a self-referencing cellular automaton is not fit to be a puzzle level. Cellular automata are also used in machine learning literature. A cellular automaton type called Classificational Cellular Automaton was used as a classifier in a Multiple Classifier System [38].

We used three games in our study. These games are a single-player maze, a multi-player maze, and Sokoban. Both mazes and Sokoban are well-studied games from various perspectives. Maze is a game where the player tries to move on to a specific tile to exit a labyrinth-like level. Maze solving is mainly studied as a path-finding problem, and according to Goyal *et al.*, two of the most effective methods are identified as the A* search algorithm and the Dijkstra's algorithm

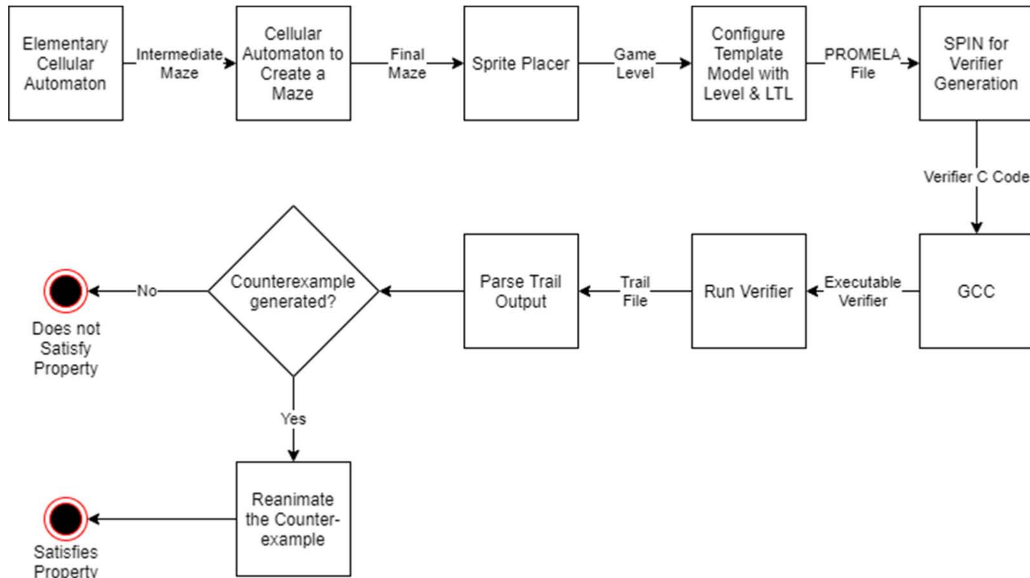


FIGURE 1. Overview of the proposed pipeline.

[39]. We used the A* search algorithm as a baseline in our research for comparing our proposed method between these two methods.

Sokoban is a game that consists of a pusher who must push several boxes into a set of designated storage locations without getting itself or the boxes stuck [7]. Sokoban is a game that is proved to be an NP-hard and PSPACE-complete problem [40], [41]. Usage of pattern databases is salvaged for finding optimal solutions to Sokoban levels by Pereira *et al.* [42]. Pattern databases store the shortest distances from abstract states to abstract goal states, and they are used to find optimal heuristic functions by introducing an intermediate goal state. Another successful method for solving Sokoban levels is the iterative-deepening A*, proposed in [43]. However, to the best of our knowledge, no study in the literature uses a model-based approach to solve any of these games.

IV. METHODOLOGY

We have developed a framework that procedurally generates a maze-like level and verifies whether it fits the requirements provided. Four different template models are proposed for the games we used in our study. These models represent the game rules in the language of the model checker. To verify whether a level is solvable or satisfies the game designer's requirement, the template game model is configured with the level-in-test. The verification output is re-animated as playable levels on GVG-AI, allowing game developers to perceive the gameplay.

The games that we used with our pipeline are:

- Maze: A single-player maze game that has only one exit point. The maze contains only walls and floor, and there are no items to interact with other than the exit point.
- Race: A maze game in which the player is racing to solve the maze against an NPC that uses an A* algorithm to find an exit point. The maze consists of walls and floors only, and there are no items to interact with except the exit point.
- Sokoban: A puzzle video game in which a player pushes around boxes in a warehouse to place them in predetermined positions. The player wins when all boxes are in their correct positions. The game has a few variants. The Sokoban variant is used in this work where the player tries to fill holes with boxes around. This is also the variant presented with PyVGDL as an example game.

The proposed pipeline in Figure 1, which is explained in detail in the following sections, works as follows: First, a maze level is generated with the help of two levels of cellular automata. The width of the level is determined from a random line that the user feeds, and the height of the level is taken equal to its width, making levels square. The line fed to the pipeline is also used as a random seed for the maze level generation. Then key sprites are placed in the maze to create a game level. Next, the template model corresponding to the game is configured according to the generated level. This configuration includes embedding the level map and sprite locations in the PROMELA code's initialization process. The requirements to be satisfied are also defined in LTL at this stage. By default, the aim is to find a winning sequence of actions, so the formula states that the game level can never be won. At this step, custom game design properties such as the player having to make turns at least N times to win the game can also be given as an LTL formula. Next, the model checker SPIN is run to emit a trail containing a sequence of moves to be performed to win the level or show how the custom property holds. Finally, PyVGDL plays these moves on the

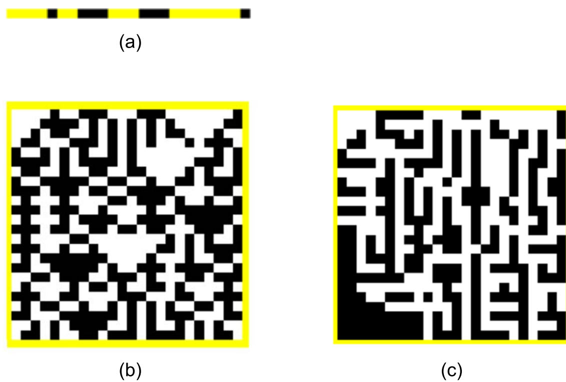

(a)
(b) (c)

FIGURE 2. **Steps of procedural maze generation. (a) The random line, (b) The output of the first level of cellular automata (c) The output of the second level of cellular automata.**

map to re-animate how the requirement is satisfied. Also, the score obtained by the gameplay is collected and displayed. The pipeline that includes a level generator and a level verifier is available at [44].

A. LEVEL GENERATOR

This phase consists of two steps: maze generation using cellular automata and sprite placement.

1) MAZE GENERATION

Our level generation module uses an elementary cellular automaton implementing rule 150, one of the chaotic rules according to Wolfram [11]. The cells can be in one of two states, a floor or a wall, being 0 and 1, respectively. We used empty cells for padding, and we started the automaton with a randomly generated line of full and empty cells. Every next state of the cells is appended below the starting line until the level becomes a square. The output looks like a maze, but it is too rough. To make it more maze-like, we applied another level of operation.

The next component of the cellular automaton is a life-like cellular automaton specialized to generate maze-like outputs [12]. The rule is abbreviated as B3/S12345 and works as follows. If a cell is a floor and has precisely three Moore neighbors marked as a wall, it turns into a wall; if a wall has one to five wall cells in its Moore neighborhood, it stays as a wall. The rule is calculated repeatedly on the maze candidate until the changes between generations are negligible. For a 24 by 24 maze, this value is taken as four or less.

Figure 2 presents the three main steps of maze generation. Figure 2-a shows a random line to start generation. Figure 2-b shows the output of the elementary cellular automaton with rule 150. As seen, the output is quite complex and contains closed rooms with sawtooth-like walls. To generate a more maze-like level, B3/S12345 rule is applied. The final output of the maze generation step is in Figure 2-c. The result is a maze-like level with lots of corridors with straight walls. Also, the output contains no closed rooms.

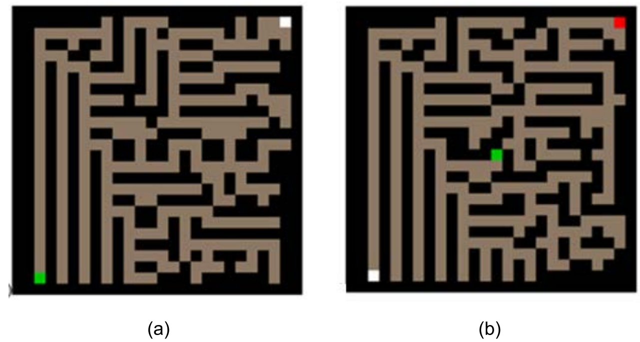

(a) (b)

FIGURE 3. **Generated maze-like levels with sprite placement (a) for maze, (b) for race.**

2) SPRITE PLACEMENT

We made different placement choices for each different game type. For the game "Maze," we place the avatar and the exit on the two-floor cells that are most far away in Manhattan distance, which is often on opposite ends of one of the diagonals of the level. For the game "Race," the avatar and the opponent are placed on the two-floor cells that are most distant, and the exit portal is placed at an equal distance to both sprites in a beeline. Since the level is not symmetrical and the opponent moves optimally, using A* search, some generated levels are unwinnable.

B. BEHAVIORAL MODELS

We developed four behavioral model templates. The first one encodes behaviors of "Maze" levels. There exists only one process for the game, representing the avatar. The avatar process includes only a main loop, and it makes a non-deterministic, navigation-based decision every iteration of the loop until the avatar reaches the exit. This decision moves in one of the four directions: up, down, left, or right. The non-determinism represents possibilities in the game play. Recall that the model checker will investigate all possibilities, including all choices made, during verification; therefore, the model does not have to keep track of which moves have been made. Since there is no rule against the total number of moves, there is no way to lose a valid maze level. However, a maze level can never be won if poorly generated or designed, having no way to reach an exit.

The PROMELA model of the avatar process is shown in Figure 4. In the code, the movement and updating of the map are inside an atomic block. Recall that this language is for parallel processes, and the keyword atomic tells the model checker to execute the statements in the block as single instruction without considering possible process interleavings. Atomic blocks make the search more efficient as it is considered one transition. This feature is used to increase the performance of the model checker for this game.

Configuring the model means adding the map initializing process according to the maze game level generated and defining LTL properties representing the game requirements. One solution produced with a configured version of the maze

```
/*The map is a 2D array where floor cells are
marked as 0, wall cells are marked as 1, the
avatar position is marked as 2, and the goal
cell with exit is marked as 3. The map is set
up in the init process that is implemented
while configuring the model.
The game ends when the avatar reaches the
exit. A Boolean variable, "win", denotes this
condition and becomes true only when the
avatar reaches the goal cell.*/
proctype avatar_mazesolver(int x; int y) {
 map[x].a[y] = 2;
// This is the main loop.
// Avatar chooses one of the available moves
until it reaches the exit.
 do
  /*Every :: mark stands for a non-
  deterministic choice. In verification
  mode, the model checker will investigate
  each choice.*/
 :: (win == 0) -> //goal is not reached yet
  /*Avatar looks for available moves in four
  directions and chooses one of them non-
  deterministically*/
  if
  ::map[x].a[y-1] != 1-> /* there is no wall
in the upper cell. move up is available*/
   atomic { /*atomic makes the model checker
to consider this block as single instruction*/
     if
     ::map[x].a[y-1]==0->
       //upper tile is a floor, move up
      map[x].a[y]= 0; //mark here as floor
      map[x].a[y-1] = 2; //avatar's new
position
      y = y - 1;
     ::map[x].a[y-1]==3-> win=1;
     /*upper tile is the goal cell. Avatar
     reached the exit denoted as win=1*/
     fi
    }
   ::map[x-1].a[y] != 1->
     //left move is available
     atomic {.../* Avatar moves left if the
     cell on its left is a floor. */
     /* Avatar reaches the goal, encoded as
     win=1, if the left tile is the goal
     cell*/
     }
```

FIGURE 4. The avatar process model for the maze game.

```
  ::map[x].a[y+1] != 1->
     //move down is available
     atomic {.../* Avatar moves down if the
     cell below is a floor.*/
     /*Avatar reaches the goal, encoded as
     win=1, if the tile is the goal cell*/
     }
  ::map[x+1].a[y] != 1->
     //right move is available
     atomic {.../* Avatar moves right if the
     cell on its right is a floor. */
     /* Avatar reaches the goal if the right
     tile is the goal cell*/
     }
  fi; //End of the move choices.
 :: else -> break /* win==1 denoting that the
 Avatar has reached the goal cell. Game ends.*/
 od;
}
```

FIGURE 4. *(Continued.)* The avatar process model for the maze game.

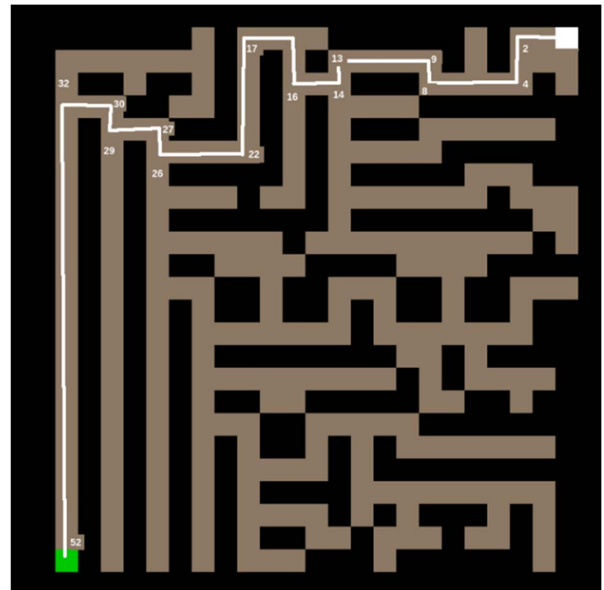


FIGURE 5. An example solution produced by the maze model. Avatar's starting position is marked with a white square. The exit is marked with a green square. The path to exit is drawn as a white path. The number indicates the number of moves done so far. For example, the avatar reached to exit in 52 moves.

model is given in Figure 5. The path to take by the avatar in this solution is highlighted with the color white, starting from the white square to the green one. The number of moves that it takes to arrive at every corner on the path is given in the figure.

The second behavioral model template is for the "Race" game levels. This model template is navigation-based. The model includes two process types, one representing the avatar and one representing the non-player character (NPC) opponent. The map of the level is shared among these two processes. The "Race" game is a turn-based game.

```
/* Two rendez-vous channels for processes to
pass turns */
chan avatar_turn = [0] of {bit};
chan opponent_turn = [0] of {bit};
/* win and lose are global Boolean variables.
win becomes true only when the avatar reaches
the exit and lose becomes true only when the
opponent reaches the exit*/
proctype avatar_race(int x; int y){
// x,y denotes the position of the avatar
 map[x].a[y] = 2; //mark the Avatar on map.
 do
   /* Game loop: Avatar waits for its turn.
   First check if the opponent has reached
   the exit that ends the game. Otherwise
   avatar makes a navigational decision,
   makes a move and passes the turn.*/
 ::((win == 0) && (lost == 0)) ->
  avatar_turn ? 0; /*avatar waiting for its
turn.*/
  if
   :: lost == 1 -> break /* the opponent has
reached the exit. Break the game loop*/
   :: else -> skip
  fi;
  if //avatar makes a move on the map
   ::((map[x].a[y-1] != 1) && ((map[x].a[y-
1] != 4)) ->
      /*The upper cell is not a wall nor the
   opponent(represented with 4)*/
    atomic {if //Avatar is going up
    ::map[x].a[y-1] == 0 ->
    //The upper cell is a floor.Regular move
     map[x].a[y] = 0;
     map[x].a[y-1] = 2; y = y - 1;
    ::map[x].a[y-1] == 3 ->
    //Upper cell is the exit. Winning move
     win = 1
    fi; }
    ::((map[x].a[y+1] != 1) &&
      ((map[x].a[y+1] != 4)) ->
       /*the cell below is not a wall nor the
       opponent. similar operations to make
       the avatar move down*/
    ::((map[x-1].a[y] != 1) &&
      ((map[x-1].a[y] != 4)) ->
```

FIGURE 6. Process type definition to model the avatar behavior in the race game.

```
       /*the cell on the left is not a wall
       nor the opponent. similar operations to
       make the avatar move left*/
    ::((map[x+1].a[y] != 1) &&
      ((map[x+1].a[y] != 4)) ->
       /*the cell on the right is not a wall
       nor the opponent. similar operations
       to make the avatar move right*/
  fi;
  opponent_turn ! 1;
       //Pass the turn to opponent
 :: else -> break  /*The game has ended.
Avatar or opponent has reached the exit.*/
 od;
}
```

FIGURE 6. *(Continued.)* Process type definition to model the avatar behavior in the race game.

To achieve a turn-based model, two rendezvous PROMELA channels are used. After making a move, the avatar process passes its turn to the opponent by sending a signal to the opponent's channel. The opponent-process receiving the signal performs a move while the avatar is waiting on the other side of its channel to get the turn. The processes pass the turn to each other after making decisions until one of them reaches the exit point. Figure 6 gives the process description of the avatar for "Race" levels. The avatar process makes non-deterministic, navigation-based decisions in every iteration of its loop, similar to the "Maze" model. Figure 7 gives the process description of the opponent. The opponent process uses the A* search algorithm. Since we use PyVGDL to replay the action sequence generated by the model checker, the opponent has to execute the same algorithm both in the model and in the re-animation of the game. Therefore, a C code snippet implementing the A* search algorithm is inserted into the process definition of the opponent.

After configuring this template with the map generated in the previous step of the pipeline, the resulting model describes possible behaviors to be produced by the turn-based race game with an A* opponent. Recall that the model checker does not select one of the non-deterministic choices but examines all possible choices using the automata-theoretic model checking algorithm given in Section II.

A solution produced by the "Race" model is in Figure 8. In the figure, the path taken by the avatar is highlighted with white, and the path taken by the opponent is highlighted with red. Also, in the figure, the number of moves that both players need to do to arrive at every corner is given.

The last two behavioral model templates are for Sokoban levels. One is a navigation-based model, while the other one only considers the moves that change a place of a box, called the push-level model. The navigation-based model is similar to the models introduced above, except this time, the model checks if moves can be used to solve a Sokoban map. An avatar process makes navigation-based decisions in

```
c_code{#include "../astar.c"};
/*Uses the same global channels and variables
as the avatar process.*/
proctype opponent(int x; int y;) {
//x,y denote the position of the NPC
 map[x].a[y] = 4; //put Opponent on the map.
 do
   /* Opponent waits for its turn. Then, first
   check if game has ended in the avatar's
   last turn. Otherwise, the opponent calls
   the A* search code to make its move. */
 :: (win == 0 && lost == 0) ->
  opponent_turn ? 1; /*opponent waits for its
turn*/
  if
   :: win == 1 -> break
       //Avatar has reached the exit
   :: else -> skip
  fi;
  c_code{/*calculating the next location
using Astar implemented in C.*/
  calculate_next_move_to_portal
    (Popponent->x, Popponent-> y,
     &(now.next_x), &(now.next_y));
    //new location is next_x and next_y.
  };
   map[x].a[y]=0; /*mark the current position
a floor cell*/
   map[next_x].a[next_y] = 4; /*mark the new
position as opponent*/
   x = next_x; y = next_y;
   if
    :: map[next_x].a[next_y] == 3 ->
       //the opponent has reached the exit
     lost = 1; break
    :: else -> skip
   fi;
   avatar_turn ! 0 //Pass the turn.
  :: else -> break //The game has ended.
 od;
}
```

FIGURE 7. Process type definition to model the opponent behavior in the Race game.

a loop and updates the map array representing the game level. Navigation can result in a basic movement or a push. The navigation-based model for solving Sokoban levels is given in Figure 9.

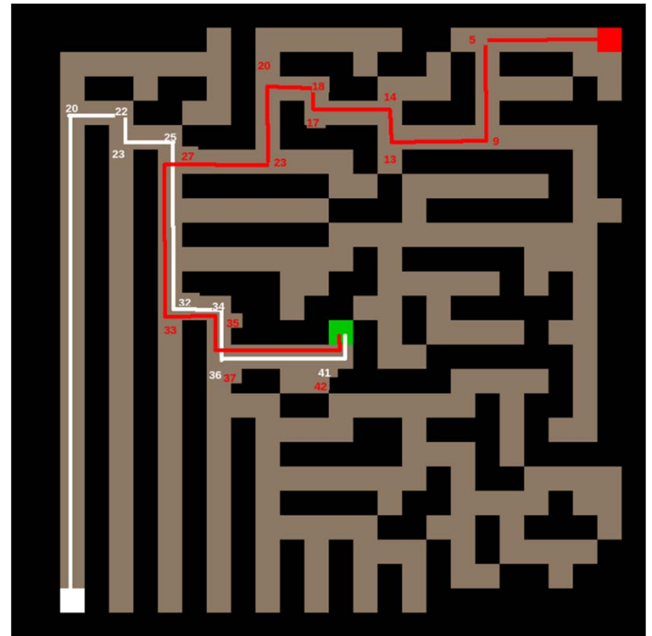


FIGURE 8. An example solution produced by using the race model. Green cell is the exit. The white cell is the initial position of the Avatar. The red cell is the initial position of the opponent. The paths of the avatar and the opponent are shown in white and red lines, respectively. Avatar reached the exit in 41 moves, whereas the opponent reached it in 42 moves.

For the navigation-based model, more challenging levels become impossible to win because of the gigantic search space of a PSPACE-complete game like Sokoban. Thus, we adopted a more sophisticated approach to this specific game presented by [7]. This model abstracts away the moves that do not change the positions of any of the boxes. Considering this abstraction, we proposed a push-level model for reducing the search space. Like the other Sokoban model, this model also defines a process to describe the avatar behavior making decisions until the game ends. However, this time, the decisions are about the pushes that the avatar can make, instead of the basic navigation moves. Before every loop, the model is informed about pushes that the avatar can perform without changing any other sprite's position beside itself, and the avatar process decides non-deterministically on doing one of these pushes. Computing available pushes on the map is performed via a helper C code. This code finds the boxes that may be pushed and looks for whether they are reachable. The avatar process gets the list of possible pushes before every decision through an array that is shared between the helper C code and the PROMELA code. After the model makes a choice, the 2D array representing the game level is updated accordingly; all the avatar moves to make the push possible were made, and the pushing move is recorded. The loop continues until there are no pushes available or there is no box left, which is the winning condition.

The expectation from push-level search is to reduce the state-space by its abstraction. In this model, any push is equally likely to be done at any point. The situation is different in the navigational model. All one-cell movements

```
/* The avatar marked as 2 in the map. Floor
cells marked as 0. Wall cells on the map are
marked as 1. Boxes on map are marked with 3,
and holes are marked with 4. */
proctype avatar_sokoban(int x; int y) {
 map[x].a[y] = 2;
//set the avatar position on the map
   /* The avatar makes navigational decision
   at each iteration of the game loop. At each
   move, if the target cell is marked as box,
   it is checked whether the box could be
   pushed. When a box is pushed into a hole,
   both the hole and the box is removed.
   The game is won when all of the boxes are
   removed. This condition is represented by
   a Boolean variable called win. The game
   loop ends when win becomes true.*/
 do
  ::(win == 0) ->
  //not all boxes are removed yet
   if
   ::map[x].a[y-1]!=1 && map[x].a[y-1]!=4 ->
   //the upper cell is not a wall nor a hole
    atomic { //Avatar decides to go up.
     if
     :: map[x].a[y-1] == 0 ->/*The upper
cell is a floor. Regular move.*/
        map[x].a[y] = 0;
        map[x].a[y - 1] = 2; y = y - 1
     :: map[x].a[y-1]  == 3 ->/*There is a
box in the upper cell. A push move.*/
      if
      :: map[x].a[y - 2] == 1 ||
         map[x].a[y - 2] == 3-> skip
         //cannot push the box up
      :: map[x].a[y - 2] == 4 ->
         //push up the box into hole.
         // Remove box and hole from the map
         map[x].a[y - 2] = 0;
         map[x].a[y] = 0;
         //update avatar position
         map[x].a[y - 1] = 2; y = y - 1;

       remaining_goals=remaining_goals - 1;
       //decrease the number of boxes
        if  //Check whether the game is won.
        :: remaining_goals == 0 -> win = 1
       //all of the boxes are removed
        :: else -> skip
        fi;
      :: map[x].a[y - 2] == 0 ->
       //push the box upwards
           map[x].a[y - 2] = 3;
           // update avatar position
           map[x].a[y] = 0;
           map[x].a[y - 1] = 2; y = y - 1
     fi
    fi  }
   ::map[x].a[y+1]!=1 && map[x].a[y+1]!=4 ->
       /*Same operations are done for avatar
       choosing to move down.*/
   ::map[x-1].a[y]!=1 && map[x-1].a[y]!=4 ->
```

FIGURE 9. Process type definition to model the opponent behavior in the race game.

```
       /*Same operations are done for avatar
       choosing to move left. */
   ::map[x+1].a[y]!=1 && map[x+1].a[y]!=4 ->
       /*Same operations are done for avatar
       choosing to move right.*/
 :: else -> break
 //The avatar has won the game.
 od;
 }
```

FIGURE 9. *(Continued.)* Process type definition to model the opponent behavior in the race game.

```
c_code{#include "../sokoban.c"};
   /*Include a helper C code that computes
   available pushes on the map. For each box,
   it checks whether the box is reachable from
   the current position of the avatar and it
   is possible to push. This code also
   provides a function to perform the pushes
   on the map. */
   /*The template process model is for a game
   level with 1 box. For every other box, four
   more elements choice will be added during
   tailoring.*/
proctype avatar_sokoban(int x; int y) {
 map[x].a[y] = 2; //put avatar on the map
 c_code{sokoban_init();};/*Initialize helper
code*/
 do
   ::(win != 1) ->  /*not all the boxes are
removed yet*/
   /*The process makes a choice from the
   choices array, and all the management of
   the map, choices, and remaining_goals are
   done by the helper C code to reduce the
   state space. After a choice has made all
   variables are updated accordingly.*/

   If //non-deterministic choices.
   :: (choices[0] == 1)-> c_code {push(0);};
   :: (choices[1] == 1)-> c_code {push(1);};
   :: (choices[2] == 1)-> c_code {push(2);};
   :: (choices[3] == 1)-> c_code {push(3);};
   fi;
   if
   :: remaining_goals == 0 -> win = 1; break
      //win the game only if no boxes left
   :: else -> skip
   fi;
 od;
}
```

FIGURE 10. Push-based model of the avatar process for the game Sokoban.

are equally likely in the navigational model, so the far away boxes are less likely to be pushed than near ones. This will result in randomly switching boxes while solving the level in the push model, and the avatar is likely to perform lots of transitions between boxes. This will make the resulting movesets to be larger in the push-based model. In the end, the expectation from the push model is to have a higher solving rate but with a larger solution size than the navigational one.

The push-level model is presented in Figure 10. The array named choices is used for communication between the model and the helper C code. The model code in this figure will

be configured according to the number of boxes in the game level by setting the size of the choices array. There are four elements in this array for each box, one for each direction. Each array element for a box encodes whether it is possible to push the box in that direction.

C. VERIFYING GAME DESIGN PROPERTIES

Every game design aims to satisfy some properties which are decided at the start. Designers mainly use a source to look up when it comes to design for guidance. One of these guides is Jesse Schell's book "The Art of Game Design" [45]. One chapter of the book is dedicated to designing puzzle games and includes a "Ten Puzzle Principles" list. Some of the principles can be related to the level design, while others are related to the design of game rules or how levels are cascaded. Since we picked three different games with predetermined game rules and checked game levels one by one, the principles of the game rules are not applicable in our case.

Four principles in this book are applicable to the level design:

#3: "Give a Sense of Progress." Although this principle can be interpreted as progress between levels, it also can be interpreted as progress in a level. Thus, it can be related to the level design.

#4: "Give a Sense of Solvability." This principle is highly related to principle #3 and can be considered a level design issue for the same reasons.

#6: "Parallelism Lets the Player Rest." This principle can be related to level design since a parallel set of challenges may exist at a level, so it should be possible to reach the same goal differently.

#9: "Give the Answer!". This principle is about giving the solution when the player gives up. This is not a level design decision, but serving the solution and ensuring the level's solvability can help the level designer.

The listed principles are all qualitative properties. A designer may have quantitative goals as well. Such goals may include the number of turning moves, straight moves, or their ratio to each other [46]. To see the effectiveness of our work, we checked both qualitative and quantitative properties to verify various game levels and explained them in the following subsections.

1) SENSE OF CONTINUOUS PROGRESS AND SOLVABILITY

Schell states that a good puzzle game should make the player see progress when solving a problem, as principle #3 [45]. This property is highly coupled with principle #4, convincing the player that the game is solvable. To demonstrate these properties, we chose Sokoban because there is an opportunity to display visual progress in this game. Since there are several boxes at the start, and the goal is to clear all of them, we decided to take the number of boxes remaining in the level to keep track of the user's progress.

To check whether a continuous progression exists in the level, we add an integer variable to the model code to monitor the number of moves performed to remove one single box. This integer variable, named c_moves, counts the number of consecutive moves and is reset when a box is removed. We define the progress property as follows: the maximum consecutive moves without visual progress should not exceed a given limit. The limit is an integer constant given by the designer and denoted by N. During tailoring, this limit is defined in the initialization process. We check that it is possible to solve this level as well. To check whether these properties are satisfied in each Sokoban level, we used the LTL formula:

$$[](!(\text{win})||(\text{c_moves} > N)) \tag{1}$$

Here win is a boolean variable that becomes true when the avatar removes all of the boxes as shown in Figure 9 and Figure 10. The formula can be translated into: "This level of Sokoban can never be won, or the consecutive moves done without visual progress is always greater than N." SPIN generates a counterexample of the tailored behavioral model with respect to this LTL property. Using this output, our framework displays a gameplay in GVG-AI that solves the puzzle where the consecutive moves done without visual progress stay under or equal to N.

Our motive to have this LTL formula is because it encodes principles #3 and #4. Counting and limiting the moves that do not make a box placed in its destination encodes the sense of progress. Having a sense of progress keeps the user interested in the game. If visual progress takes much time, users can quit a game due to a lack of interest. Also, seeing progress increases the sense of solvability. If visual progress in a puzzle level takes too much time, a user can quit playing since this situation may make the user judge the level as too hard to solve. By including these two principles of Schell, this LTL formula can be used to show that a game level holds a requirement of constantly grabbing the player's attention.

2) PRODUCING A SOLUTION

Principle #9 requires giving the solution to the level when the player gives up [45]. So, a good framework that checks a game level has to present a winning sequence of actions to the player or designer. We use the LTL formula [](!win), which states that the variable win will never be true, to make the model checker generate a winning sequence of actions as a counterexample. However, to generate that 'Aha!' moment in a puzzle game, we need to present an understandable and straightforward way to victory, mostly the shortest one. To produce the shortest path to victory, we used a run-time option of SPIN to fetch the counterexample with the smallest state transitions. In our case, this means the one with the least moves.

After the counterexample with the shortest path is found and recorded to a trail file, the trail file is played back to re-animate to capture which moves are done to achieve victory in the level. After getting the moves performed in the winning sequence, with the help of PyVGDL, the counterexample is

played visually in a window to show the solution produced by our framework to the game designer.

This formula encodes principle #9 with a very small size. As the complexity of the verification algorithm is exponentially related to the formula's size, this LTL formula can be used to verify a game level in an environment with fewer resources as fast as possible.

3) POSSIBILITY OF DIFFERENT SOLUTIONS

Schell states that there should be more than one way for a player to finish a puzzle game successfully, as in principle #6 [45]. Having more than one way to finish the game corresponds to generating more than one winning sequence. If at least two different sequences of actions can be generated as a counterexample, we accept that the level is solvable with more than one strategy. We need SPIN to generate two counterexamples with different state transitions to display this property. Our framework implementing the pipeline can instruct SPIN with run-time options to achieve this. After SPIN generates more than one counterexample in multiple trail files, the tool checks the filesystem for the trail files and decides whether the level is open to different strategies or not.

We use the same LTL formula of [] (!win) we used for principle #9 to produce a winning sequence to demonstrate these properties. We pass the LTL formula with our models to SPIN and run with the "-c2" option to force it not to stop when it finds the first violation but to continue the search to find another one. If there is more than one solution, we conclude that the level can be won with different strategies.

This formula and the run-time option encode Schell's 9th principle and help check a game level against a requirement of increased solvability, which is stated in principle #6. Having more than one solution lets the player choose one of the possible solutions to win. This may result in a player solving a level easier as it does not enforce the player to discover the single unique solution.

4) NUMBER OF TURNING MOVES, AND NUMBER OF STRAIGHT MOVES

Instead of qualitative design goals, the level designers may have quantitative requirements to fulfill. Kim *et al.* state that the number of turning moves or number of straight moves done to win a maze can be desired properties while designing a maze [46]. To check these properties, we modify the models to keep the count of turning and straight moves performed to finish the game. We add two integer variables to the models: where t_moves represent the number of tuning moves and s_moves represent the number of straight moves. Then, we use the following two LTL formulae to represent these quantitative requirements:

$$[](!(\mathrm{win})||(\mathrm{t_moves} > N)) \tag{2}$$

$$[](!(\mathrm{win})||(\mathrm{s_moves} > N)) \tag{3}$$

These formulae are aimed to produce a trail where the game is won, and the number of turning/straight moves performed is less than or equal to the desired number given by a game designer.

Maze designers may desire such a property since having too many straight or turning moves may bore the player, and some sort of balance may be required between such movements.

5) TWISTINESS OF THE PATH TO THE VICTORY

The twistiness of a path is the ratio of turning moves to straight moves. An increased twistiness might require the player to be more agile to win. Kim *et al.* state that turning moves' ratio to the straight moves can be a desired property when designing mazes [46]. To state such properties, we add two counters, t_moves and s_moves into the game model as discussed above. Then, the twistiness property is specified using the following LTL formulae:

$$[](!(\mathrm{win})||(\mathrm{t_moves}/\mathrm{s_moves} < N)) \tag{4}$$

$$[](!(\mathrm{win})||(\mathrm{s_moves}/\mathrm{t_moves} > N)) \tag{5}$$

These two LTL formulae encodes the requirement of having a specific ratio of turning moves against straight moves in the winning path.

V. EXPERIMENTS

In this paper, three different games are used with the GVG-AI framework. Two of these games are implemented for this study, while one is used as-is. The implemented two games are simple maze games. In the game "Maze", an avatar is expected to solve a maze with no other sprites involved. In the game "Race," the avatar is expected to solve the maze before its opponent does. The last game is Sokoban, a well-known game for its computational difficulty [41].

The experiments are grouped into three main categories. The first group of experiments is done to determine the level sizes for the other experiments. The second group of experiments compares the solution generation performance of the proposed model-based approach against existing methods. There are three different experiments in this group. We first compare the proposed model-based approach against the A* search algorithm. Next, the proposed approach's performance is compared against Monte Carlo Tree Search in the game "Race." The last experiment in this group compares two different models proposed for Sokoban levels. Since Sokoban is a demanding game to solve, this experiment aims to show how optimizations on a model may affect performance.

The third group of experiments demonstrates the ability of our approach in the verification of game design properties. There are two different experiments presented in this category. The first one is to verify levels for a qualitative property, while the second is to verify levels for a quantitative property (see Section V). The first experiment in this category aims to check "Race" levels for the existence of multiple

ways to victory. The other experiment aims to check maze levels for the twistiness of the path to victory, which is defined as the turning moves' ratio to the straight moves [46].

All experiments are done on a computer that has an Intel Core i5-3470 CPU at 3.2 GHz clock frequency, 16 GB of RAM, and CentOS 7 as its operating system. All the experiments were run in a single core of the CPU. More in-depth explanations of the experiments are given in their respective sub-sections.

A. PRELIMINARY EXPERIMENTS FOR DETERMINING PARAMETERS

We needed to decide on the level sizes to use for the upcoming experiments. EXPERIMENT 1 is performed to determine this parameter. This experiment is designed as follows. Three different candidates will be used in the subsequent experiments: A* search, the proposed Model-Based approach, and Monte Carlo Tree Search. We create a maze level 8×8 and give it to all three techniques to solve. Then we generate new levels by increasing the level size two by two until one of the candidates fails to deliver a solution.

For MCTS, we have established a time limit that is 100 times more than the time spent by our model-based approach. The last level size played will be designated as the maximum level size for the rest of the experiments.

The metric collected in this experiment is the success rate of the candidates with respect to different sizes.

B. EXPERIMENTS FOR PERFORMANCE COMPARISONS

1) EXPERIMENT 2

Our second experiment aims to see the scalability of the proposed model-based approach with respect to one of the popular path-finding algorithms, the A* search. To assess the performance of the proposed approach versus the A* search, we used the navigation-based model for the game "Maze" given in Figure 4. To generate the shortest path as its output, we used SPIN with the option that returns the counterexample with the least number of state changes. Since the model is navigation-based, the state changes occur in every move done by the avatar. Thus, the model-based approach is instructed to produce the shortest path possible when this option is used. The performance of our approach is compared with the performance of an agent implemented in PyVGDL employing an A* search. As the heuristic of the A* search, Manhattan distance is used.

We have given both candidates the same maze levels in nine different sizes, all generated by the pipeline described in Section IV.A. We generated 50 different levels for two candidates to solve for each size. There are 450 levels in total.

The metrics we have collected for the experiment are the average solving time and the average solution length with respect to level size.

2) EXPERIMENT 3

The third experiment aims to compare the performance of the proposed approach with Monte Carlo Tree Search (MCTS). MCTS is a heuristic search algorithm notable for its employment in games. The advantage of MCTS is its usability without domain knowledge: The state information, possible actions, and the results of these actions are sufficient to use MCTS in any game.

MCTS may waste computational time with back-and-forth movements without progress. In this experiment, therefore, we modified the game environment to record how many times a tile is moved on and gave a movement penalty to avoid MCTS being stuck. Since repetitive movements will result in more penalties, the agent is encouraged to explore further in the maze faster instead of doing repetitive actions.

Two competitors, our approach and MCTS, were run in two different games, "Maze" and "Race." Both solvers are fed with the same levels in three different sizes generated by the pipeline we proposed. Both candidates are given the same 50 levels to solve (300 levels in total). MCTS's exploration parameter 1.41 (square root of 2) is chosen, and MCTS agent is given a hundred times more time to complete its search because MCTS has a disadvantage. This disadvantage is employing no domain knowledge, and the proposed model-based approach is built on domain knowledge. The average length for the solutions found and the solving rate are collected as metrics.

3) EXPERIMENT 4

The fourth experiment compares the performances of the two proposed template models for Sokoban. These two models are the navigation-based and push-level models presented in Section 4. These two competing models are run in the same 20 levels from Alberto Garcia 1-1 level pack [10].

While the navigation-based model works at a more detailed level, resulting in a larger state-space, the push-based model is more abstract and optimized for a smaller state-space with a performance loss on solution optimality. The main expectation from the experiment is to get higher solution rates with the push-level model while sacrificing the shortness of solution lengths. The experiment shows the effects of optimizations such as abstractions or simplifications on complex and resource-hungry problems.

The metrics compared in this experiment are the average time competitors take in a level, the number of levels solved, and the solution length.

C. EXPERIMENTS ON VERIFYING COMPLEX PROPERTIES

1) EXPERIMENT 5

The aim of this experiment is to show how the proposed method can be used to check a level with respect to a qualitative requirement. The requirement chosen here is the level having multiple ways to victory, as discussed in Section IV.C.

The game "Race" is played against a bot opponent using the A* algorithm to make optimal moves. Since the player has no obvious advantage against its opponent in the sprite placement, some of the levels end up unwinnable. Also, some

of the winnable levels end up having a one-move margin between the avatar and its opponent to win. In addition to having a solution, Schell states that a puzzle should have more than one way for a player to finish successfully [45]. Therefore, we used this game in the experiment to check the requirement of having more than one solution.

For this experiment, 200 "Race" levels with the size of 24 × 24 are generated with the pipeline presented in Section 4. The number of levels that satisfy the requirements is kept as a metric.

2) EXPERIMENT 6

This experiment aims to show how the proposed model-based approach can be used to verify quantitative requirements. The requirement chosen is the twistiness of the solution path, as explained in Section IV.C. Using our framework, 200 "Maze" levels with the size of 24 × 24 are generated. The levels are checked against the requirement of having a minimum twistiness of 20%. Since SPIN can generate counterexamples that have more twistiness with repetitive actions, a hard limit is placed on the number of moves done while solving a level. This limit is chosen as 55 moves, being ten more from the average shortest path in a 24 by 24 maze, according to the results found in Experiment 3. The metric collected in this experiment is the number of levels that satisfy the requirement.

VI. RESULTS AND DISCUSSION

In this study, we have focused on the following questions:

- How does the model-based approach compare to the A* search algorithm on scaling in shortest path finding?
- How does the model-based approach perform in puzzle games with respect to Monte Carlo Tree Search?
- Does the model-based approach scale with a game like Sokoban, which is known to be hard to solve in limited memory or limited time?
- How can the model-based approach be employed to verify qualitative and quantitative requirements?

A. EXPERIMENT 1

Experiment 1 was conducted to determine the maximum level size to be used in the experiments. The experiment continuously increased the level size until one of the three candidates could not solve any level. The metrics collected here are the solving rate of three candidate-level solvers. We collected this metric to compare the candidates and determine a maximum level size for the upcoming experiments.

Table 2 shows the result of this experiment. It gives the percentage of levels solved by the candidates with respect to the level size. The results indicate that levels larger than 24 by 24 are not fit for MCTS. The larger levels are impractical to distinguish the model-based approach from the MCTS since the rate of solving for the MCTS will not change for larger levels. Also, larger levels do not distinguish the model-based approach from the A* search since they are both 100% successful.

TABLE 2. Results of experiment 1. Solving rate of three candidates on varying level size.

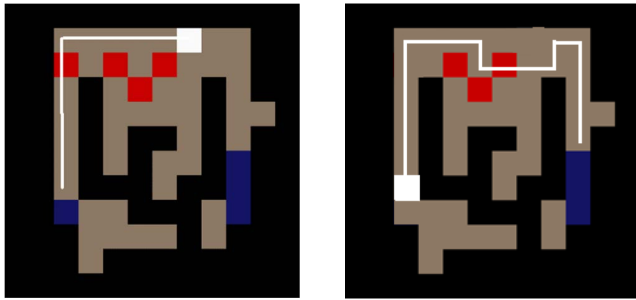
Level Size	Model-Based	A-Star Search	MCTS
8x8	100%	100%	40%
10x10	100%	100%	37%
12x12	100%	100%	31%
14x14	100%	100%	25%
16x16	100%	100%	18%
18x18	100%	100%	15%
20x20	100%	100%	9%
22x22	100%	100%	4%
24x24	100%	100%	0%

B. EXPERIMENT 2

Experiment 2 aim to answer the first question. We used our template model proposed for the game "Maze" against the classical A* search for this performance comparison. The metrics collected here are the average time for solving a level and the average size of the solution brought up with respect to both competitors' level sizes. The main reason for measuring the average time to solve is to see the performance of the model-based solution with respect to the algorithmic solutions in path-finding problems. However, a direct comparison would not be comparable since the environments of the two are different. The model-based solution has multiple file operations, a source compilation, execution of this compilation's output, and a Python code moderating all of these steps, while the A* search algorithm was implemented in Python. So, we have normalized these datasets to see how the solving time increased while the level size increased. As a result, the information extracted from this metric is not a direct performance comparison but a scalability comparison. The second metric is included to show the ability of the model-based solution in the shortest path finding.

Figure 13 shows the average time measured for the candidates to solve levels with respect to the levels' size. However, since the question is on scalability, we normalized both data with respect to their first data point. Thus, the data represents how much the average time for solving a level increased while the level size increased. Also, this normalization neutralizes the differences between the two candidates. Figure 14 represents the very same data, without normalization.

Figure 14 shows that the performance of the model-based approach is catching up with the A* path-finding algorithm. The figure shows that the proposed approach is more scalable. With the increasing level size, the A* search seems to lose more performance relatively than the proposed approach. When the total space is multiplied by 9, the proposed methodology's average solution-finding time only increased 75%, while its competitor's increased nearly 70 times.



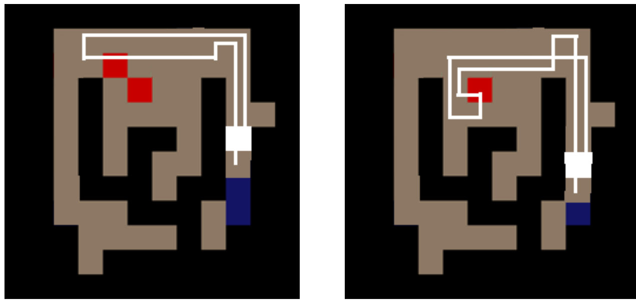
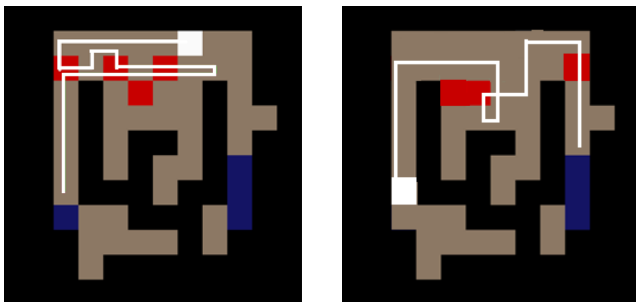


FIGURE 11. An example solution created by the navigation-based model. Each image shows movements to put a box into its place. For example, Avatar moves left in the top left image until it reaches the leftmost box and then pushes it to the bottom.



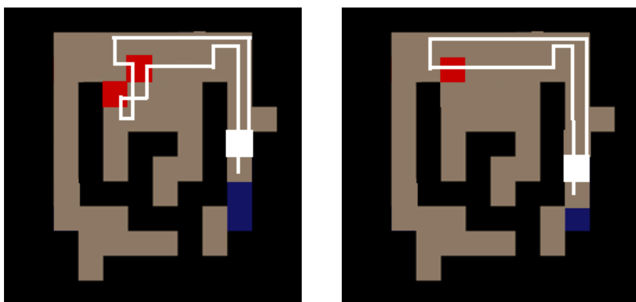


FIGURE 12. An example solution created by the push-based model. Avatar is white; boxes are red, and target places for boxes are blue. Each image shows pushes and paths to put one box in its place. For example, in the top left image, Avatar pushes the leftmost box down, the second leftmost to down, and the rightmost box to the right. Then goes all the way left and pushes the first box to the bottom.

In answer to our first research question, even if the A* search is faster for smaller levels than the proposed method, the trend in data shows that it is fair to assume that this performance gap will close as levels get larger. Figure 14 shows that the two candidates become comparable with the level size of 18, and the competition becomes close with the size of 24.

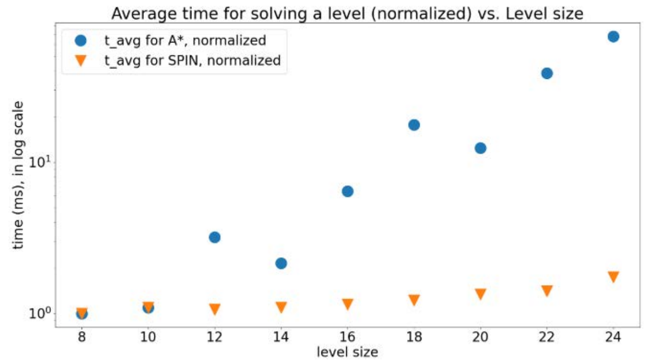


FIGURE 13. Experiment 2 results comparing proposed method with A-star search in terms of average time to solve a level with respect to level size, normalized.

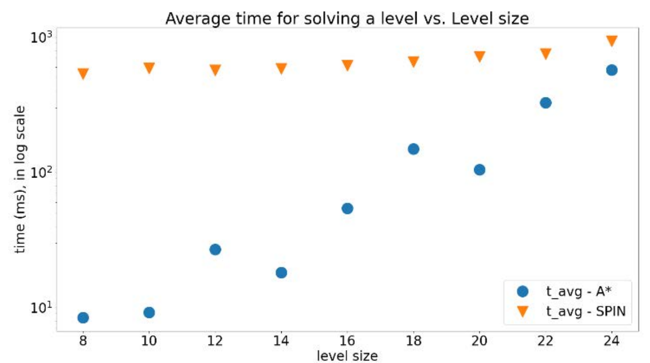


FIGURE 14. Experiment 2 results comparing the proposed method with A-star search in terms of average time to solve a level with respect to level size.

The results of this experiment indicate that the model-based approach can be used as an alternative to an algorithmic approach in path-finding problems. Since the proposed approach includes multiple file operations, using the algorithmic approach on smaller levels would be better.

C. EXPERIMENT 3

Experiment 3 aims to find an answer to the second question. To assess the performance of the model-based approach with respect to MCTS, two different games are played in three different sizes by two candidates. The metrics collected here are the rate of success for both candidates and the average length of the solution they brought up in two different games and three different sizes.

Tables 3-6 display the results of the experiment. Table 3 shows the success rates of both candidates in three different sizes for the game "Race." Table 4 shows the same metric for the game "Maze." Table 5 shows the average solution length of both candidates in three different sizes for the game "Race." Table 6 does the same for the game "Maze."

These results show that for maze solving, the success rate of the MCTS drops drastically as the level size grows. For

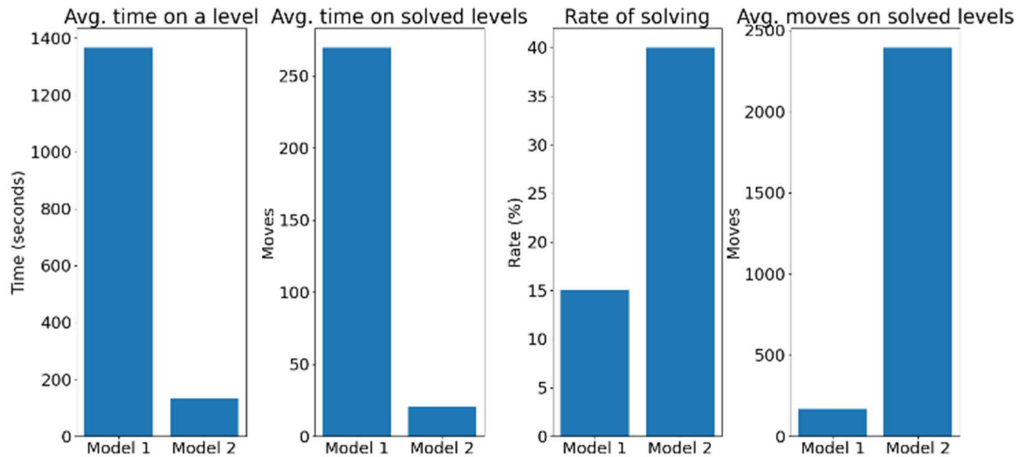


FIGURE 15. Metrics collected from Experiment 4. Model 1: Navigational, Model 2: Push-based.

TABLE 3. Success rates of MCTS and model-based approach on the game race.

Level Size	Success Rate MCTS	Success Rate Model-based
8x8	89%	100%
16x16	21%	100%
24x24	0%	100%

TABLE 4. Success rates of MCTS and model-based approach on the game maze.

Level Size	MCTS	Model-based
8x8	34%	100%
16x16	14%	100%
24x24	6%	100%

the game "Race," since the game is not forgiving against the player's mistakes, referring to the sprite placement section, the drop in the solve rate of the MCTS is even sharper. Although the performances of both methods are close for the smallest levels, in all six cases, the model-based approach is more performant than its competitor and opens the gap as the level size increases.

Our second research question indicates that the proposed approach has a specific advantage over MCTS in maze-like puzzle games since it relies more on computational power and memory; while the input size increases, the gap between the two competitors' performances grows even more.

While doing these experiments, our expectations were parallel to the results we got from the experiment. We were aware of the advantage the model-based approach has over the MCTS. MCTS is a non-informed search, while the model-based approach is built on domain information. Nevertheless, even if the two approaches are not exact alternatives to each other, the results point out a practical advantage of using a model-based approach over MCTS.

TABLE 5. Average solution length for the game race.

Level Size	MCTS	Model-based
8x8	6.75	6.11
16x16	20.33	16.14
24x24	Not solved	27.06

TABLE 6. Average solution length for the game maze.

Level Size	MCTS	Model-based
8x8	42.12	13.54
16x16	127.14	27.72
24x24	353	37.72

D. EXPERIMENT 4

Experiment 4 aims to answer the third question about the scalability of the model-based approach on Sokoban. The navigation-based and push-based models introduced in Section IV.B are used to solve 20 Sokoban levels of [10] to achieve this.

The metrics collected are the average time spent on a level, the success rate of solving, and the average solution length for both models in the same Sokoban levels. The main expectation was for the push-based model to have faster and more solutions to the levels but with longer solutions. The reason for this expectation is that by abstracting away the moves that do not change a box position, the push-model reduces the state-space of the problem. Therefore, more levels should have become solvable with this abstract model. We were expecting higher solution rates with the push-based model by reducing the memory costs. Also, since the state-space is reduced, it should take less time to search that space. So, the expectation was to have significantly shorter times to solve a level. However, since the push-based model is more abstract than the other, we expected more inefficient solutions and more comprehensive solutions on average. We expected the navigation-based model to have

a lower success rate because the system would run out of memory. Figure 15 shows the result of this experiment.

Figure 15 shows that a push-based model increased the solving rate and decreased solving time by nearly 90% compared to a navigation-based model. However, the average length of solutions has increased by 15 times. We believe further optimizations on the push-based model can reduce the resulting move set, such as tunnel macros or goal macros [7]. To address our third research question, this experiment shows that template models can be subject to optimizations in the model design to get a better result if needed. Also, it shows that SPIN's different options that enable different optimizations can be used to improve performance.

The results indicate that the model-based approach may react differently when certain decisions are made. In the experiment, that decision was shrinking the search space using the push-level search, which trades the optimality of the solution with the performance and the speed.

E. EXPERIMENT 5

Experiment 5 aims to answer the fourth research question with a qualitative requirement. The game "Race" was chosen to conduct the experiment. The experiment was done to check procedurally generated levels for the game "Race" with respect to the requirement of having multiple ways to finish the game. The data collected here are the number of winnable levels and the number of levels that have an alternate way of winning the level.

Three factors affect the result of this experiment. First, the avatar or its opponent has no obvious advantage over each other in the level generation phase. Either one can be advantageous, or none of them may be. Second, the player has the advantage of the first move. Both in the model and the game, the player moves first, before its opponent. This gives the player an advantage in neutral levels in the level generation phase, where the goal is equally distant to the player and its opponent. The last one is that having an alternate way to victory can be ensured by either having more than one equally long but different path or having an advantage over the opponent that can afford to make a sub-optimal move. In the light of these factors, our expectation from the experiment was as follows: We were expecting that more than half of the levels generated would be winnable with the advantage of the first move of the avatar. Also, we were expecting that the levels generated which satisfy the requirement would be just under but near 50%.

Out of 200 Race levels generated in the experiment, 113 of them were winnable—the model-based approach found in 95 out of these 200 levels more than one way to victory. The result that we obtained from the experiment looks parallel with the expectations. Since the avatar has the advantage of the first move, 56.5% of the levels generated end up being winnable. This result is fitting to the expectation of having more than half of the levels being winnable. 47.5% of the levels generated have an alternate path of victory for the avatar, which is aligned with the expectation of having less than half of the levels having an alternate path. This result indicates that the proposed model-based approach can be used to verify qualitative requirements to answer our fourth question.

F. EXPERIMENT 6

Experiment 6 aims to answer the fourth research question with a quantitative requirement. The game of choice to verify a quantitative requirement is the game "Maze". The requirement chosen to check levels is having at least 20% twistiness which is the ratio of turning moves done by the avatar to all moves performed. Since twistiness can be increased by making a turning move repeatedly, we established a limit to the total moves that can be done. This limit is chosen with the help of the result from Experiment 2. Since the average shortest path in a 24 by 24 maze was nearly 45 in Experiment 2, the upper limit for the number of total moves is chosen as 55. This limit is because the requirement can be met by performing repetitive movements. We have determined this value to prevent SPIN from abusing the game rules to check the level.

The data collected here is the number of levels that satisfy the requirement. We expected nearly all of the levels to be verified in this experiment. There are two main reasons for this expectation. First, the target percentage is relatively low. For a target placed at the cross corner of the level, 20% twistiness seems achievable. One regular turn makes four straight moves acceptable, and one U-turn makes eight straight moves acceptable. The other reason is the upper limit for the number of total moves. For an average level, the avatar has ten extra moves that can be used to boost the twistiness of the winning path. Because of these two reasons, we were expecting a tiny percentage of levels not satisfying this requirement.

In the experiment, 196 out of 200 generated Maze levels satisfy the requirement. This result is in the same direction as the expectations. With the help of low target and extra moves, 98% of the levels generated end up satisfying the requirement. This result also indicates that the proposed model-based approach can be used to verify quantitative requirements to answer our fourth question.

VII. CONCLUSION

This paper aims to provide a model-based level verification framework for maze-like puzzle games. To that end, we presented a pipeline that begins with puzzle development using two-level cellular automata, followed by an automated configuration of the pre-developed model templates of these games, which is then run using the model checker program SPIN re-animated on GVG-AI. This re-animation enables the game developers to perceive the gameplay readily with the help of visual cues. The proposed technique was tested against the A* path-finding algorithm in a maze solving game and a slightly modified Monte Carlo Tree Search algorithm in a game with a low tolerance for sub-optimal actions. In addition, two different models of the game Sokoban were

run against each other to study the trade-offs taken to scale a model for a memory-intensive game.

Our results indicate that the suggested method's mean performance converges to one of the top path-finding algorithms in a simple maze game and significantly beats Monte Carlo Tree Search in a different game with an opponent. Furthermore, our findings demonstrate that the proposed strategy may be adjusted or enhanced by making trade-off considerations to scale for more difficult-to-solve games. Although this research focuses on maze-like puzzle games, it may be applied to various different genres.

In the future, we aim to increase our performance by employing more complicated linear temporal logic specifications, probabilistic model checkers, and use cloud computing to improve our study. In addition, we will expand our model pool by incorporating models for different games.

APPENDIX

In this appendix we give the output of the model checker SPIN as well as original counterexample generated at the verification phase.

After configuring the template file with respect to a Sokoban level, we instructed SPIN to generate a verifier using the following command spin -a temp.pml. The generated verifier C code is compiled with the following instruction:

```
gcc -std=c99 pan.c -DMAX_LEN=11
-DNUM_OF_BOXES=2 -DNOFAIR -DBITSTATE
-DNOBOUNDCHECK -o.. /spin/temp.
out -lm
```

The output of the verifier is 2214 lines. The verification result is at the end of the output. The result is as follows:

pan:1: assertion violated !(!(!(win))) (at depth 1216)
#Level got verified
pan: wrote temp.pml.trail
#The path to verification is at temp.pml.trail.
(Spin Version 6.4.6 – 2 s 2214 December 2016)
Warning: Search not completed #Because of -c1
+ Partial Order Reduction
Bit statespace search for:
never claim + (ltl_0)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)
State-vector 224 byte, depth reached 1591, errors: 1
5925 states, stored
1422 states, matched
7347 transitions (= stored+matched)
0 atomic steps
hash factor: 22652.8 (best if > 100.)
bits set per state: 3 (-k3)
Stats on memory usage (in Megabytes):
1.424 equivalent memory usage for states (stored*(State-vector + overhead))
16.000 memory used for hash array (-w27)
0.076 memory used for bit stack
0.534 memory used for DFS stack (-m10000)
16.925 total actual memory usage

The counter-example generated by the verifier is dumped into a trail file. The trail file is replayed with the command./temp.out -r -s temp.pml.trail to make it human readable. The result of the replay is 197 lines. The content of this file is as follows:

MSC: ∼G 3
Push@ 3 5; Side: 0
Push@ 3 7; Side: 0
Push@ 4 5; Side: 2
…..//continues for another 23 lines
…..//this is the beginning of the counterexample. Line27
pan:1: assertion violated !(!(!(win))) (at depth 1217)
spin: trail ends after 1217 steps
#processes 3:
1217: proc 0 (ltl_0) temp.pml:3 (state 6) (invalid end state)
(!(!(win)))
(1)
1217: proc 1 (:init:) temp.pml:147 (state 4)
-end-
1217: proc 2 (avatar_sokoban) temp.pml:138 (state 35) (invalid end state)
printf('Won\n')
global vars:
byte remaining_goals: 0
bit win: 1
bit map_inited: 1
//…continues for 152 lines
local vars proc 2 (avatar_sokoban):
int x: 2
int y: 2
int last_move: −2
//end of file.

This output is hard to process for a user. Our pipeline parses this file and reanimate the gameplay accordingly.

REFERENCES

[1] O. Drageset, M. H. M. Winands, R. D. Gaina, and D. Perez-Liebana, "Optimising level generators for general video game AI," in *Proc. IEEE Conf. Games (CoG)*, London, U.K., Aug. 2019, pp. 1–8.
[2] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An automated model based testing approach for platform games," in *Proc. ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst. (MODELS)* Ottawa, ON, Canada, Sep. 2015, pp. 426–435.
[3] L. Mugrai, F. Silva, C. Holmgard, and J. Togelius, "Automated playtesting of matching tile games," in *Proc. IEEE Conf. Games (CoG)*. London, U.K., Aug. 2019, pp. 1–7.
[4] P. García-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo, "Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone," *Knowl.-Based Syst.*, vol. 153, pp. 133–146, Aug. 2018.
[5] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Proc. 13th Haifa Verification Conf.*, Haifa, Israel, Nov. 2017, pp. 99–114.
[6] O. Tekik, "Verifying maze-like game levels with model checker SPIN," M.S. thesis, Grad. School Inform., Middle East Tech. Univ., Ankara, Turkey, 2021.
[7] F. Marocchi and M. Crippa, "Monte Carlo tree search for Sokoban," Tesi di laurea Magistrale, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico Di Mila, Milan, Italy, Tech. Rep., 2017.

[8] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms," *IEEE Trans. Games*, vol. 11, no. 3, pp. 195–214, Sep. 2019.
[9] T. Schaul, "A video game description language for model-based or interactive learning," in *Proc. IEEE Conf. Comput. Inteligence Games (CIG)*, Niagara Falls, ON, Canada, Aug. 2013, pp. 1–8.
[10] A. Garcia. *Alberto Garcia 1–1 Sokoban Level Set*. Accessed: Apr. 16, 2022. [Online]. Available: https://www.sokobanonline.com /play/web-archive/alberto-garcia/1–1
[11] S. Wolfram, "Universality and complexity in cellular automata," *Phys. D, Nonlinear Phenomena*, vol. 10, nos. 1–2, pp. 1–35, 1984.
[12] D. Eppstein, "Growth and decay in life-like cellular automata," in *Game of Life Cellular Automata*. London, U.K.: Springer, 2010, pp. 71–97.
[13] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[14] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency* (Lecture Notes in Computer Science), vol. 1043. Berlin, Germany: Springer, 1996, pp. 238–266.
[15] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.
[16] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Proc. Int. Conf. Protocol Specification, Test. Verification*. Boston, MA, USA: Springer, 1995, pp. 3–18.
[17] L. Feng, D. Chen, H. Lonn, and M. Torngren, "Verifying system behaviors in EAST-ADL2 with the SPIN model checker," in *Proc. IEEE Int. Conf. Mechtron. Autom.*, Aug. 2010, pp. 144–149.
[18] N. Goga, S. Costache, and F. Moldoveanu, "A formal analysis of ISO/IEEE P11073–20601 standard of medical device communication," in *Proc. 3rd Annu. IEEE Syst. Conf.*, Mar. 2009, pp. 163–166.
[19] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Boston, MA, USA: Addison-Wesley, 2008.
[20] R. Vereecken. *PyVGDL*. Accessed: Apr. 16, 2022. [Online]. Available: https://github.com/rubenvereecken/py-vgdl
[21] O. Tekik. *PyVGDL, Forked*. Accessed: Apr. 16, 2022. [Online]. Available: https://github.com/iamonur/py-vgdl
[22] C. Baier and J. P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.
[23] I. Hasegawa and T. Yokogawa, "Formal verification for node-based visual scripts using symbolic model checking," *IEICE Trans. Inf. Syst.*, vol. 105, no. 1, pp. 78–91, 2022.
[24] N. Igawa, T. Yokogawa, M. Takahashi, and K. Arimoto, "Model checking of visual scripts created by UE4 blueprints," in *Proc. 9th Int. Congr. Adv. Appl. Informat. (IIAI-AAI)*, Sep. 2020, pp. 512–515.
[25] S. Radomski and T. Neubacher, "Formal verification of selected game-logic specifications," in *Proc. Eng. Interact. Comput. Syst. SCXML*, Berlin, Germany, Jun. 2015, pp. 30–34.
[26] L. T. Holloway, "Modeling and formal verification of gaming storylines," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. Texas Austin, Austin, TX, USA, 2016.
[27] A. Yacoub, M. E. A. Hamri, and C. Frydman, "DEv-PROMELA: Modeling, verification, and validation of a video game by combining model-checking and simulation," *Simulation*, vol. 96, no. 11, pp. 881–910, Nov. 2020.
[28] W. Kavanagh, A. Miller, G. Norman, and O. Andrei, "Balancing turn-based games with chained strategy generation," *IEEE Trans. Games*, vol. 13, no. 2, pp. 113–122, Jun. 2021.
[29] R. Rezin, I. Afanasyev, M. Mazzara, and V. Rivera, "Model checking in multiplayer games development," in *Proc. IEEE 32nd Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, May 2018, pp. 826–833.
[30] E. Marques, V. Balegas, B. F. Barroca, A. Barisic, and V. Amaral, "The RPG DSL: A case study of language engineering using MDD for generating RPG games for mobile phones," in *Proc. Workshop Domain-Specific Modeling (DSM)*, Oct. 2012, pp. 13–18.
[31] P. Milazzo, G. Pardini, D. Sestini, and P. Bove, "Case studies of application of probabilistic and statistical model checking in game design," in *Proc. IEEE/ACM 4th Int. Workshop Games Softw. Eng.*, May 2015, pp. 29–35.
[32] P. Moreno-Ger, R. Fuentes-Fernández, J.-L. Sierra-Rodríguez, and B. Fernández-Manjón, "Model-checking for adventure videogames," *Inf. Softw. Technol.*, vol. 51, no. 3, pp. 564–580, Mar. 2009.
[33] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proc. Workshop Content Gener. Games*, Jun. 2010, pp. 1–4.
[34] C. Adams and S. Louis, "Procedural maze level generation with evolutionary cellular automata," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Nov. 2017, pp. 1–8.
[35] A. Pech, P. Hingston, M. Masek, and C. P. Lam, "Evolving cellular automata for maze generation," in *Proc. Australas. Conf. Artif. Life Comput. Intell.*, Feb. 2015, pp. 112–124.
[36] Y. P. A. Macedo and L. Chaimowicz, "Improving procedural 2D map generation based on multi-layered cellular automata and Hilbert curves," in *Proc. 16th Brazilian Symp. Comput. Games Digit. Entertainment (SBGames)*, Nov. 2017, pp. 116–125.
[37] T. P. Pavlic, A. M. Adams, P. C. W. Davies, and S. I. Walker, "Self-referencing cellular automata: A model of the evolution of information control in biological systems," 2014, *arXiv:1405.4070*.
[38] P. Povalej, P. Kokol, T. W. Družovec, and B. Stiglic, "Machine-learning with cellular automata," in *Proc. 6th Int. Symp. Intell. Data Anal.*, Madrid, Spain, 2005, pp. 305–315.
[39] A. Goyal, P. Mogha, R. Luthra, and N. Sangwan, "Path finding: A* or dijkstra's?" *Int. J. IT Eng.*, vol. 2, no. 1, pp. 1–15, 2014.
[40] D. Dor and U. Zwick, "SOKOBAN and other motion planning problems," *Comput. Geometry*, vol. 13, no. 4, pp. 215–228, Oct. 1999.
[41] J. C. Culberson, "Sokoban is PSPACE-complete," Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, Tech. Rep. TR 97-02, 1997.
[42] A. G. Pereira, M. R. P. Ritt, and L. S. Buriol, "Finding optimal solutions to Sokoban using instance dependent pattern databases," in *Proc. 6th Int. Symp. Combinat. Search*, Jun. 2013, pp. 141–148.
[43] A. Junghanns and J. Schaeffer, "Domain-dependent single-agent search enhancements," in *Proc. 6th IJCAI*, Stockholm, Sweden, 1999, pp. 570–577.
[44] O. Tekik. *Maze-Like Game Levels Verification With SPIN, Pipeline Repository*. Accessed: Apr. 16, 2022. [Online]. Available: https://github.com/iamonur/the_legendary_pipeline
[45] J. Schell, *The Art of Game Design: A Book of Lenses*. Boca Raton, FL, USA: CRC Press, 2008.
[46] P. H. Kim, J. Grove, S. Wurster, and R. Crawfis, "Design-centric maze generation," in *Proc. 14th Int. Conf. Found. Digit. Games*, Aug. 2019, pp. 1–9.

ONUR TEKIK received the B.S. degree in electrical and electronics engineering from Middle East Technical University (METU), Ankara, Turkey, in 2016, where he is currently pursuing the M.S. degree with the Graduate School of Informatic' Information Systems.

His research interests include procedural content generation and software engineering.

ELIF SURER (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Boğaziçi University, Turkey, in 2005 and 2007, respectively, and the Ph.D. degree in bioengineering from the University of Bologna, Italy, in 2011.

She is currently working as an Associate Professor with the Graduate School of Informatics' Multimedia Informatics, Middle East Technical University (METU), Ankara, Turkey. Her research interests include serious games, virtual/augmented reality, human and canine movement analysis, machine learning, and computer vision.

AYSU BETIN CAN received the B.S. degree in computer engineering from Middle East Technical University (METU), Ankara, Turkey, in 1999, and the Ph.D. degree in computer science from the University of California, Santa Barbara, in 2005.

She is currently working as an Associate Professor with the Graduate School of Informatics' Information Systems, METU. Her research interests include design for verification, reliable concurrent software development, interface-based modular verification and specification, web services, and software engineering.