Prototyping Games using Formal Methods

Sebastian Krings[1[0000−0001−6712−9798]] and Philipp Körner[2[0000−0001−7256−9560]]

[1] Competence Centre for Information Security, Niederrhein University of Applied Sciences, Mönchengladbach, Germany
sebastian.krings@hs-niederrhein.de

[2] Institut für Informatik, Heinrich-Heine-Universität, Düsseldorf, Germany
p.koerner@uni-duesseldorf.de

Abstract. Courses on formal methods are often based on examples and case studies, which are supposed to show students how to apply formal methods in practice. However, examples often fall into one of two categories: First, many are artificial and thus do not relate to practice. Second, examples are based on projects of industry partners and therefore often are too involved for students to understand them.
In this paper, we present a different approach. By formalizing the rules of commonly known games, we achieve examples both engaging and suited for students. Furthermore, we broaden the horizon of formal methods, driving research at the same time: we present extensions such as playable visualizations and explore the relationship between game AIs and model checking heuristics.

1 Introduction

Rather than purely focusing on the mathematical foundations, courses on formal methods are often based on examples and case studies, supposed to show students how to apply formal methods in practice. However, the examples used are often quite artificial and do not relate to practice. At the same time, examples based on projects of industry partners are rooted in practice but often are way too involved for students to understand.

In this paper, we present a different approach, relying on games as examples for formal models. The models discussed are used in teaching and have been developed by students both during courses and theses. We deem games particularly suited as teaching examples for two reasons:

1. The games we use are well-known to the students. We can thus focus on the modeling and proving as well as on methodology, rather than having to discuss intended properties of our models. Essentially, reducing the amount of requirements engineering we have to perform by using common examples allows us to focus on the formal method itself.
2. Modern computer games are among the most sophisticated examples of software systems. Due to the high complexity, implementations of game semantics, e.g., rules of movement, can often only be tested scarcely and are thus naturally suited for applying formal methods.

As a simple example, consider the board game checkers. If an implementation somehow allowed moving a piece onto a white field, no valid moves would be defined afterwards. Arbitrary successor states might occur, or the game might never be over, as no other piece can capture it or vice versa. If the invalid move is only seldom possible, it might not be hit be testing procedures.

In the following, we will use the B-method [1] and its successor Event-B [2]. Both represent state-based formal methods used for modeling software and systems and proving their correctness. Models written in B or Event-B can be animated and model checked using ProB [26,25,24]. Additionally, we discuss the tools used in the development of our prototypes in Section 3.

Apart from teaching, using formal methods to prototype games has several advantages for game development itself: First, creation of a working prototype is often faster in B, due to the high level of abstraction[3]. Furthermore, step-wise refinement allows focusing on certain parts of a game. In consequence, our case studies contribute both to teaching and research.

2 A Primer on B and the B-Method

The formal specification language B [1], its successor Event-B [2] and the B-method [1] follow the correct-by-construction approach. Their models consist of a set of machines, which itself contain constants and variables together with corresponding type definitions. A predicate (which might have multiple solutions) is used to describe the initial states.

Different means of composing machines are available. Furthermore, the B-method heavily relies on abstraction, i. e., the step-wise refinement of very abstract machines towards more concrete implementations. In addition to machines, Event-B features contexts, supposed to hold static information.

Machine operations (or events, in case of Event-B) are used to specify transitions between states. A machine operation has a unique name and consists of B substitutions defining the state after execution. An operation can have a precondition allowing or prohibiting execution based on the current state. Operations can be non-deterministic and might be nested. Furthermore, B features a multitude of different substitutions, including if-then-else constructs and while loops. Event-B's events are considerably simpler and can only include guards for execution and variable assignments.

To ensure correctness of a specification, the user can define machine invariants, i. e., safety properties that have to hold in every state. Depending on the tool used, these properties can be verified either by formal proof or using model checking. In addition, LTL properties can be specified to verify temporal behavior.

Besides using the types explicitly provided by the B language specification, one can introduce used-defined types in the form of sets. A set is defined by a unique name and may be initialized by a finite enumeration of distinct elements.

[3] Regarding the trade-off between ease of implementation and efficient execution see [16] for a general point of view and [22] for a perspective on B and Event-B.

Sets not defined by enumeration are called deferred sets and are assumed to be non-empty and finite.

3 Software Used

Both for our courses and the case studies presented below we rely on three tools for development and verification of formal models. Each supports different verification techniques, such as model checking and proof. When writing specifications in Event-B, one often combines all of them instead of using only a single verification tool. Consequently, integrations into one another have been developed. In detail, the tools we use are:

- ProB [26,25,24], a constraint solver, model finder and model checker for the B family of languages. One of the key features of ProB is fully automatic animation of specifications, i. e., the user can traverse the state space without having to supply values for variables or parameters. In addition, ProB incorporates different model checking techniques, including explicit state and symbolic ones [17]. Both model checking and animation are driven by a backend written in SICStus Prolog [6], relying mainly on its constraint logic programming based solving library [7]. The Prolog kernel is supported by integrating SMT solvers [18] and SAT solvers [31] via Kodkod [33]. ProB supports LTL model checking using a tableau-based algorithm as outlined in [30,11].
- Rodin [3] is an IDE for Event-B implemented on top of Eclipse. It features generation of proof obligations, e. g., for invariant preservation, and can be combined with different provers for discharging them. In particular, one can connect the Atelier-B provers [8] and SMT solvers [12,13]. Rodin does not directly support visualization of models. Instead, ProB is provided as a plugin [5].
- BMotionWeb [20,21] is a tool for the rapid creation of formal prototypes on top of B and Event-B machines. While ProB supports basic visualizations of formulas, individual states and the state space, more involved visualizations and software prototypes are realized using BMotionWeb. In particular, it allows to link a graphical user interface to a formal specification animated by ProB.

Below, we present three case studies in which we applied the formal approach of [20] to prototyping games. In Section 4, we present a prototype of Pac-Man, while in Section 5, we are modeling chess. Afterwards, in Section 6, a version of Lightbot is presented. The case studies outline the broad applicability of formal methods to games: with Pac-Man, we have a game featuring continuous and simultaneous movement. In contrast, chess represents turn-based board games where players move in succession. Finally, the Lightbot game presents how stack-based programming languages can be implemented.

4 Pac-Man

As a first case study, we use the well-known classic arcade game Pac-Man. We will start with the set of requirements to be verified in our model in Section 4.1. The requirements are posed to the students in the same way, e.g., as a practical specification task.

The Event-B model[4] is discussed in Section 4.2. On top of the model, we used visualization techniques to implement an interactive and playable prototype as discussed in Section 4.3. In the background, the model checker drives a simple artificial intelligence controlling the ghosts as described in Section 4.4.

4.1 Requirements

We try to keep the set of requirements simple and easy to grasp to help students focus on applying the formal method rather than spending time on implementation or specification details. Furthermore, we abstract further from the original Pac-Man: Instead of being continuous, movement is made discreet, i. e., Pac-Man and ghosts move on a grid of fields.

rq1 Pac-Man can only be moved from one field of the grid to a direct neighbor field. This implies that it cannot jump to another position in the level.
rq2 Each ghost can only be moved from one field to a direct neighbor field.
rq3 Pac-Man can only be moved when every ghost, that must have been started, has moved at least once after the last movement of Pac-Man.
rq4 Pac-Man can be moved through a tunnel.
rq5 The first two ghosts must start before Pac-Man starts.
rq6 The third / fourth ghost must start as soon as 30 / 180 dots are collected.
rq7 Each dot can only be collected once.
rq8 If Pac-Man and a ghost are on the same field, one must catch the other.
rq9 If a ghost catches Pac-Man, the player loses a life.

4.2 Model and Refinement Hierarchy

The model of Pac-Man is split into different refinement levels, each introducing additional detail to the game. The first refinement level adds Pac-Man's movement, the second deals with collecting as well as scoring dots and the third and fourth refinement levels consider moving the ghosts and hunting.

Pac-Man's Movement Initially, on the first refinement level, we specify the movement of Pac-Man, focusing on requirements containing constraints immediately applicable to movement: **rq1** and **rq4**.

The maze in which Pac-Man moves is encoded as a set containing the coordinates of each field. We use two variables to store Pac-Man's position: one for its

[4] A full version of the model can be found at: `https://github.com/pkoerner/EventBPacman-Plugin/tree/master/eventb`

current and one for its prior location, used to prevent Pac-Man from jumping. Both are members of the set of fields in the maze as stated using type invariants inv101 and inv102 shown below. Combined, the invariants state that Pac-Man never leaves the maze.

After storing the prior position, an additional invariant inv103a can be used to prohibit jumping over squares. This is sufficient to verify **rq1**. Using LTL rather than state invariants, the requirement could also be checked without introducing an additional variable.

inv101: $pos \in \boldsymbol{maze}$ // current position
inv102: $prior_pos \in \boldsymbol{maze}$ // last position
inv103a: $(\text{prj}_1(pos) - \text{prj}_1(prior_pos) \in \{2, -2\} \wedge \text{prj}_2(pos) = \text{prj}_2(prior_pos)) \vee$
$(\text{prj}_2(pos) - \text{prj}_2(prior_pos) \in \{2, -2\} \wedge \text{prj}_1(pos) = \text{prj}_1(prior_pos))$

Additionally, the model includes two Boolean variables indicating if Pac-Man was moved in the last event and if it was moved at all. The latter will allow us to verify **rq5** later on. Furthermore, we can already specify that Pac-Man may move through a tunnel to the opposite side of the grid. This is done by introducing another boolean variable, which indicates whether the last movement was through the tunnel. We add this variable as a guard to invariant inv103a resulting in inv103 to allow jumps in case of tunnel traversal. We also state that if the tunnel is used, Pac-Man has to appear on the other side in inv107.

inv106: $tunneled \in \text{BOOL}$ // last movement was through tunnel
inv103: $(moving = \top \wedge tunneled = \bot) \Rightarrow$
$((\text{prj}_1(pos) - \text{prj}_1(prior_pos) \in \{2, -2\} \wedge \text{prj}_2(pos) = \text{prj}_2(prior_pos)) \vee$
$(\text{prj}_2(pos) - \text{prj}_2(prior_pos) \in \{2, -2\} \wedge \text{prj}_1(pos) = \text{prj}_1(prior_pos)))$
inv107: $tunneled = \top \Leftrightarrow \{pos, prior_pos\} = \boldsymbol{tunnel}$

An example of an Event-B event concerned with movement is shown below.

Event move_up ⟨ordinary⟩ $\widehat{=}$
any
p
where
grd101: $p \in \boldsymbol{accesible}$
grd102: $(\text{prj}_1(position) = \text{prj}_1(p)) \wedge (\text{prj}_2(position) - \text{prj}_2(p)) = 2$
grd103: $position \in \boldsymbol{tunnel} \Rightarrow p \notin \boldsymbol{tunnel}$
then
act101: $position_before := position$
act102: $position := p$
act103: $moving := \top$
act104: $tunneled := \bot$
act105: $moved := \top$
end

Collecting Dots In a second refinement, we introduce how collecting dots and thus scoring points works. The corresponding requirements are **rq6** and **rq7**. In this step we also add the four super pills which enable Pac-Man to catch ghosts. Furthermore, we add distinct events for moving Pac-Man to an empty field or to a dot.

The first three invariants introduced in this refinement state that the score is a natural number (inv201 and inv202) and, furthermore, that it can only be increased by either 10 or 200 points or remain the same value (inv203).

inv201: $score \in \mathbb{N}$ // current score
inv202: $score_before \in \mathbb{N}$ // last score
inv203: $score - score_before \in \{0, 10, 200\}$ // possible values for increment
inv204: $scoredots_current \in \mathbb{P}(scoredots)$ // uncollected small dots
inv205: $ghostdots_current \in \mathbb{P}(ghostdots)$ // uncollected super pills
inv206: $counter_scored = \mathrm{card}(scoredots) - \mathrm{card}(scoredots_current)$ // dots collected

Invariants inv204 and inv205 give type information for the collectible dots. Additionally, we count the amount of dots collected in *counter_scored*. Then, inv206 ensures that no dots get lost, i. e., that every collected dot is awarded to the player's score.

The Ghosts, Lives and Hunting After movement and scoring, a third refinement is used to add the ghosts. This includes their movement as well as catching Pac-Man. Variables, invariants and events corresponding to the ones we added for Pac-Man's rules of movement are added for each ghost as well.

This refinement step satisfies **rq2** in the same way **rq1** was implemented earlier. Additional invariants are used to ensure **rq6**. Again, we can check **rq5** with an LTL formula: for all possible paths through the state space, the ghosts have to move before the Pac-Man can start. In another refinement step we introduce the hunting of ghosts activated when Pac-Man collects a super pill: once collected, all ghosts are added to a set of currently catchable ghosts. Once caught, they are removed from this set. Additional guards are added to the movement events to ensure only huntable ghosts can actually be caught. Otherwise, the ghost will catch Pac-Man. Furthermore, we added the lives the player has and that he loses one if he gets caught. Invariant inv402 ensures that Pac-Man cannot gain an infinite amount of lives. Moreover, inv404 implies that Pac-Man cannot gain an additional life and only can lose at most one life at a time. Furthermore, in this refinement step, we can check both **rq8** and **rq9** using LTL formulas.

inv402: $lives \leq \boldsymbol{start_lives}$ // current lives
inv404: $lives_old - lives \in \{0, 1\}$ // at most one life is removed
inv406: $chased_ghosts \subseteq \{\boldsymbol{ghost_1}, \boldsymbol{ghost_2}, \boldsymbol{ghost_3}, \boldsymbol{ghost_4}\}$ // hunted ghosts

The last refinement step adds further movement rules. These rules enforce the order of movement, e. g., that the second ghost moves after the first. Additionally, to satisfy **rq3**, Pac-Man can only be moved once all ghosts have been moved.

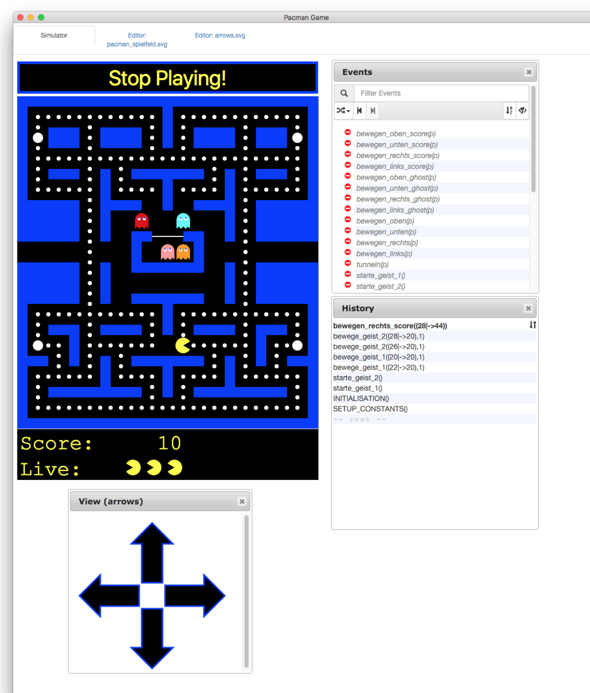

Fig. 1. Pac-Man Visualization

4.3 Visualization

After the model is initialized, the visualization shows the maze and Pac-Man as well as the ghosts in their starting positions. In addition, the score value is shown together with the three small Pac-Mans representing the lives of the player. Afterwards, the visualization reacts to changes of the model using the observer pattern of BMotionWeb, i. e., we register observers for the variables and define how the elements of the visualization react to state space changes.

Usually, events are executed by the user by selecting them from a list of enabled events. To get closer to a playable prototype, we added four arrow buttons. We registered all events to move the Pac-Man in a direction to each of the corresponding arrow buttons. When the user clicks on one of the buttons, BMotionWeb executes the event if it is enabled in the current state.

Additionally, we implemented a listener for key events, enabling the user to play the model just like the real game by using arrow keys.

4.4 Adding a Simple Game AI on Top of Formal Models

Pac-Man also served as a playground for two novel research directions:

- Can the prototypical model made playable without further code generation, i.e., by executing it directly via the model checker. In particular, we were interested to see how user interaction could be designed and what level of responsiveness could be reached. This line of research later lead to a general implementation of runtime usage of B models [19].
- Furthermore, Pac-Man allowed us to experiment with state-space search algorithms beyond simple depth-first or breath-first traversal.

To gain a test-bed for both questions, we use the Groovy API of BMotionWeb in order to implement a simple AI that is able to control the Pac-Man and the four ghosts. It supports three different modes of operation: First, if the user plays with the arrow keys, the AI lets Pac-Man move in the given direction until it hits a wall and moves the ghosts each tick. Secondly, the user may click the Play!-button and the AI plays the game against itself. This means the Pac-Man and the ghosts are completely controlled by the AI. Lastly, the AI can be used in order to move the ghosts automatically after the user moved the Pac-Man.

In the first two cases, we run a loop in a thread until the user stops it or the game is over. It moves both the Pac-Man and each of the ghosts. While in the first case, the Pac-Man simply follows its current direction, in the second case the AI decides where Pac-Man should turn. As a heuristic, we use a breadth-first search in order to find the nearest dot to score. This directly corresponds to the search strategy used in the underlying model checker.

In the last case, the model checkers search strategy is only used to control how the four ghosts are moved. After a specific operation is executed by the used controlling Pac-Man, we again use breadth-first search to identify possible paths for the ghosts.

5 Chess

As a second case study, we implemented the well-known board game chess in B[5]. Again, we will discuss our model and the set of requirements and continue with visualization and the integration of a game-playing AI in our model.

5.1 Requirements

When posed as a specification task to students, we usually provide the following set of requirements, leading to a prototype that can be considered playable:

rq1: Pieces can only be moved in their specific way (e. g., a king can only move exactly one field into any direction).
rq2: If the king is in check, only moves getting the king out of check are permitted.
rq3: No piece can be moved outside the 8×8 board.
rq4: Special moves (Castling, En Passant and Promotion) follow the rules.

[5] The main B machine can be found at:
`https://github.com/pkoerner/b-chess-example/blob/master/b/board.mch`

rq5: If the king cannot be defended immediately, the game is lost.
rq6: If no legal move is possible for one player, the game is considered as a draw.
rq7: Both players have the same set of pieces and the white player has the first move.

5.2 Model and Refinement Hierarchy

Rather than relying on refinement as done with the Event-B specification of Pac-Man, we use the modularization capabilities of classical B and split our model into a model containing the board, another for visualization and a third containing basic variables and sets.

There are two different ways to specify the board: A *piece-centric* approach associates all pieces with the field they occupy, e. g., $white\ king \rightarrow e1$. In contrast, a *square-centric* approach maps each field on the board to the piece on it. This could be done using a partial function (to avoid mapping empty files to placeholders) or using a total function (which can be beneficial for constraint solving and visualization). In this case study, we opted for a square-centric representation using a total function. We accept the corresponding overhead in order to find empty fields more easily.

Movement Moving pieces is encoded using B operations, i. e., each move results in a state transition. Four different B operations are introduced: movement for black and white pieces each and taking a piece, again with individual operations for black and white. Special moves, such as *castling*, *En Passant* or *promoting a pawn* are added to the model in further refinement steps. Their preconditions share common predicates, checking if the figure/field combination exists, if a movement path is feasible and if the player is not in chess.

`grd_tuple`: $x \in dom(board)$ // Field x exists in board and it maps to a white or black piece
`grd_check`: $not_in_check(new_board)$ // The player is not in check after the move

Furthermore, operations have to distinguish between *moving* and *taking* a piece: When taking a piece, it suffices to check whether the movement is valid, i.e., according to the rules and all fields in between are empty. Simply *moving* a piece, however, requires an additional precondition to check whether the target field is empty.

`grd_move`: move_white_piece(piece,x,y,take,board) // move respects movement rules
`grd_fields`: $\forall field \in between(x, y).free(field)$ // fields on the way and target are empty
`grd_take`: $take = 1 \Rightarrow board(y) =$ opponent_piece // if taking: there is an opponent piece

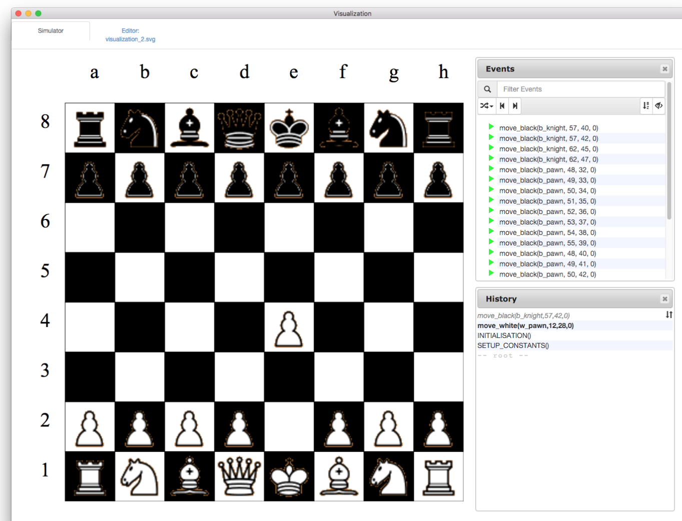

Fig. 2. Chess Visualization

Check, Checkmate and Draw In order to implement check, one needs to look one step ahead to find out if an opponent piece could take the king. This impacts performance, as for every possible move every possible opponent move might have to be calculated twice: first, when checking whether the move should be enabled or not and again after executing the operation. Additionally, we decided to encode checkmate as an invariant violation. One of the invariants claims that one white king and one black king are part of the match at all times. If one of them is taken, the invariant is violated and model checking stops.

A draw can be reached in various ways. If both players agree, the game could be declared as a draw at any time. Further, if 50 moves have passed without moving a pawn or taking a piece, it leads to If the situation on the board is deadlocked and the same position is reached too often, the game is declared as a draw. While the number of moves can be tracked using an integer variable, keeping track of all prior positions leads to a combinatorial explosion, effectively rendering model checking impossible.

5.3 Visualization

To visualize the chess board and let the user play, we again rely on BMotionWeb. On the left side of Figure 2, the visualization itself is placed. Clicking on a piece and a target field triggers the corresponding operation.

An operation can be evaluated manually by clicking on one of those listed inside the events window on the right-hand side. The history of executed events is shown below, the user can return to a former state by clicking on one of

the operations listed there. By doing so, the trace rolls back to state after the operation was executed.

5.4 Minimax as Model Checking Heuristic

Minimax is a game-independent algorithm, i. e., its implementation only differs in the game-specific evaluation functions used to determinate a value for each leaf node. For chess, we could consider the number of pieces left for each player or the value of own pieces compared to opponent pieces (e. g., a queen is more valuable than a pawn). Furthermore, one could evaluate the number of reachable fields or movability.

While using as much information as possible in general leads to a stronger AI it also renders computing the evaluation function more difficult. As a tradeoff, we decided on the following information and weights:

- Values of figures residing on the board following the valuation by Shannon E. Claude [32].
- Number of pawns in desired positions, e.g., passed pawns as well as number of pawns in undesired positions, e.g., doubled pawns.
- The number of semi-open files, i.e., the number of rows or columns the player's rooks can move at least five fields into one direction on. This is a measure rock movability, which indicates how well players can bring their rooks into play. We multiply the measure with a weight of 2.
- We count how well the fields adjacent to the own king are guarded, again applying a weight of 2.
- We measure to what extent a player controls the four squares in the center of the field. As they are usually crucial to winning the game, we apply a weight of 3.

To prevent the model checker from running too long, a relatively small *search depth* is set. Essentially, this is done by performing a depth-limited exploration of the state space and applying the evaluation function of Minimax to the reached states. The highest ranking states are then explored further, effectively driving the model checker along the path a depth-restricted Minimax would have taken. As commonly done in chess engines, paths which might yield a better situation but are too long are not considered further. At the same time, the value of a state is only influenced by the best movements the opponent can make, i.e., Minimax implicitly follows the best strategy of both players and thus is not influenced by good states that have a single bad successor state.

Since a lost game results in an invariant violation, we can now use model checking to find a playing strategy. Using ProB we try to find a path leading to a checkmate and, in consequence, a win. Due to the inherent combinatorial complexity of chess, state spaces are usually too large to be explored exhaustively. In the future, we want to study the effect of state space reduction techniques such as partial order reduction on the performance of game prototypes.

6 Lightbot

Lightbot[6] is not as universally known as Pac-Man or chess: It is an educational puzzle game with the aim of programming a robot such that it follows a specific path through a grid. On its way, it has to light up several tiles of the grid to reach the overall objective.

The instruction set to control the robot is fairly small, i.e. moving and turning the robot, letting the robot jump upwards or downwards, toggling special fields and calling specified procedures. However, this small instruction set is sufficient to implement recursive programs as well as loops and builds a Turing-complete language.

While the two former case studies have been created during Bachelor and Master theses, we have used Lightbot as a mandatory assignment in our course on safety critical systems several times. Usually, students had to specify a formal model of Lightbot including a (playable) visualization to be allowed to take the final exam. In particular, we required the model to be parametric, in the sense that it should be possible to add and change the robot's programming during execution. More general, this implies that students had to specify a model of the interpreter of Lightbot's programming language.

6.1 Requirements

The rules of the game can be best explained in form of requirements:

rq1: The robot moves on a three-dimensional board.
rq2: The game is generic, i.e., different levels (boards) are supported and can be provided and switched in some way.
rq3: The robot supports all moves (forward, toggle light, left/right turn, jumping and entering one of two sub-procedures).
rq4: The robot starts execution in the main-procedure.
rq5: A program stack is required to execute the user-defined sub-routines, as the may be mutually recursive. Again, this underlines the idea that students do in fact specify the internal workings of an interpreter.
rq6: The lowest elevation level is 1.
rq7: Starting position and the tiles the robot has to light up to complete the level are described in the level itself, not hard-coded in the interpreter.

6.2 Refinement Hierarchy

As we expect our students to follow the formal modeling process as a whole, we do not provide a particular refinement hierarchy upfront.

Our reference specification[7] however starts with modeling a two-dimensional grid that the robot moves upon. In that stage, moving up, down, left and right

[6] `https://lightbot.com/`
[7] Available at: `https://www3.hhu.de/stups/models/fmfun19/lb.zip`

is allowed if the robot faces the corresponding direction. It is also possible to turn the robot, and to light up specific tiles.

The second refinement steps adds a third dimension. This adds two different aspects to the game. First, simple movement is now blocked in case the elevation of the adjacent square is different. Second, a new kind of movement is introduced as the robot has to jump in order to move vertically. This refinement level completes the basic execution engine. It is possible to execute all enabled commands whenever one likes, with no constraints concerning a program counter or limited amount of memory.

The third refinement level is used to introduce the actual programming of the robot. We use an Event-B context to describe the level (elevation and tiles to light up), as well as the starting position of the robot and the direction it faces. Additionally, the context is used to constrain how large individual procedures implemented by the player may be.

The corresponding Event-B machine specifies how programs are specified and executed, i.e., the program has to be written beforehand and, upon interpretation, only operations at the current program counter may be executed. Additionally, a program stack is added that stores the program counters once sub-procedures are called and resumes execution upon returning.

6.3 Visualization

As with chess and Pac-Man, the current state of the game can be immediately identified when looking at a visualization instead of pretty prints of the underlying data structures. The visualization in Figure 3 also relies on BMotionWeb.

On the top left, the grid with the robot and tiles that are to be switched on (blue) and the ones already lit (yellow) is shown. Underneath, all available commands are given next to each other. Again, the game is fully playable using the visualization, e.g., one can select a procedure to add instructions to and modify it at will. The current code of the main procedure and all sub-procedures is given below.

The history Event-B events executed by the model checker (both during the construction and execution of the program) as well as all events that can be executed in the current state are shown on the right-hand side. As with the original game, once the robot begins executing the player-given code, only two Events are still permitted to be executed: fetching the next instruction and executing it.

Our reference specification and its visualization can be used in order to explain the game to the students, before they have to implement it on their own. As it is much easier to reason about a concept that one is familiar with rather than something given from an informal text-based representation, this assists students a lot in the early phase.

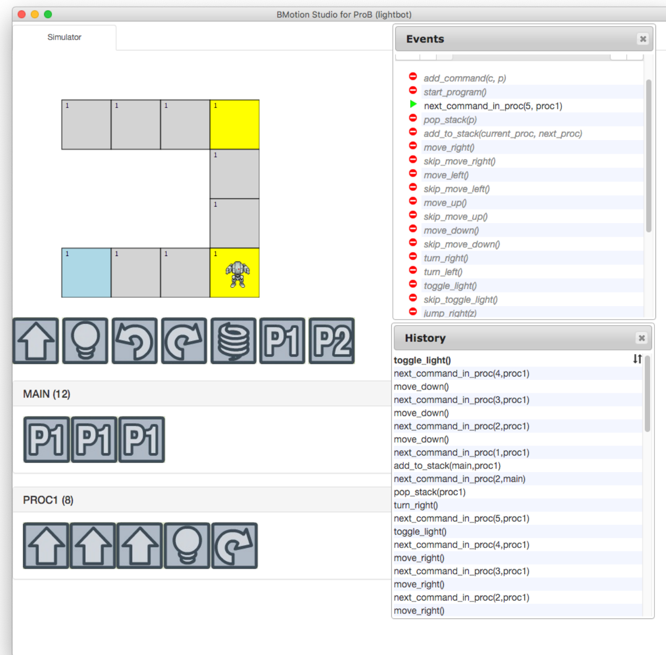

Fig. 3. Lightbot Visualization

6.4 Models of Virtual Machines

The original game is an educational game on coding. It is used in order to teach basic programming concepts, such as function calls, recursion and loops.

Following this idea, writing a specification of the game itself (as opposed to a specification of the player-given code to solve a level) teaches the same aspects on a meta-level, i.e., how to *model* and *verify* function calls, recursion and loops. Thus, students learn how to model programming languages and their interpreters.

The same concept could later be applied to "real" programming languages with more sophisticated semantics.

7 Related Work

Teaching formal methods concepts by relying on card games and card tricks rather than artificial examples has been considered by Curzon and McOwan [9]. They discuss numerous tricks and small games that visualize the concept of invariants, etc.

In his dissertation [27], Timo Nummenmaa already considered implementing game prototypes using formal software development techniques. Both in the dissertation and in the related publications [28,29] the possible impact of using formal methods for game development are discussed. In particular, the authors especially mention the benefit of executable formal models, as we provide by the combination of B and ProB. We were able to extend upon the former work thanks to BMotionWeb: we can provide richer and more interactive visualization, closer to the intended game design itself.

Formal verification of properties of checkers has been considered in [4]. The authors encode the game as a finite state system and search for winning strategies using symbolic model checking. In contrast to our work, the focus is on properties of the game itself, rather than creating playable prototypical implementations. However, [4] underlines that state spaces of (albeit simple) board games can be handled by current model checkers.

In [15], the author uses the HOL4 theorem prover [14] to verify chess endgame databases. To do so, an encoding of possible moves similar to the one in Section 5 is used. Instead of using a model checker to find and evaluate possible moves, the correctness of predefined move sequences given in endgame databases is verified.

Instead of using verification techniques to encode games, [10] considers the opposite way: Verification tasks are encoded as games, that could later be solved by people unaccustomed to software verification. Software and security constraints are represented by a simple puzzle-like game, which solution represents either failure or successful verification.

Directed model checking using different heuristics has been considered in the context of ProB in [23]. Comparable to the approach used in Section 5, the authors use state properties to control which state ProB's model checker expands next. However, heuristics are not as involved as the Minimax algorithm employed in this paper.

8 Conclusions and Future Work

In summary, our three case studies have shown both advantages and shortcomings of the tools introduced in Section 3:

- ProB (or any model checker with animation capabilities) is very important during development. (Bounded) model checking of the specification usually gives fast feedback about the correctness of a specification or an implementation. Animation, in particular with an added visualization on top, allows reassuring a developer that changes made to the specification behave as intended. Sometimes, the tool cannot cope with the entire state spaces though: e.g., assumptions about chess based on the rules cannot be model-checked, as the state space is way too large.
- Student feedback concerning Rodin is rather negative: while it provides a type checker, a proof obligation generator and proof system with some automated proof rules, usability is lacking. Sometimes, Rodin is in an inconsistent state where, e.g., POs are not generated as they should be and a

cleaning mechanism has to be invoked. Also, as Event-B files are not plain text, structural editors are default. Many students find it uncomfortable to switch between text boxes in the IDE rather than navigating with arrow keys. Furthermore, some functions such as removing certain elements are hidden in context menus that only pop up when right-clicking on very specific positions. Finally, the files do not integrate well with version control systems such as git.
– BMotionWeb is a great tool in order to explain specifications to domain experts or students, once the visualization and the model are complete. An application based entirely on web technologies proved to be hard to use though. When errors occur, it is not clear in which layer the cause is located in: is it an error in the B model? Is an SVG file broken? Is the config file incorrect? Is there a bug in the JavaScript code? As some errors are not reported, development can be cumbersome if one is not an expert in all technologies that are used by BMotionWeb.

When applied to the development of game prototypes, they support using classic formal proof and model checking to verify the correctness of game implementations. In particular, we have proven both high-level properties about the game's implementation itself and the correct representation of the rules of a game.

As we mentioned in Section 4.2, playability of game prototypes is limited, because it is hard to achieve continuous movement in an Event-B model. Nevertheless, they make for easy to understand and highly motivating examples for students trying to work their way into formal methods. Turn-based games however, such as chess or Lightbot, are a great match for "slower" execution due to interpretation overhead and can be fun and engaging to interact with.

Using BMotionWeb, we have the possibility to animate and visualize our prototypes. As we have shown, BMotionWeb is able to produce playable prototypes of both real-time and round-based games. However, the visualization behaves quite slow and is thus not usable in presence of time limits. While this is less critical for teaching and for implementing board games like chess, it limits the applicability of our approach to games in general.

8.1 Impact on Student Learning

It is hard to measure the influence on how interest, attention and understanding is enabled for students. There is no clear trend that correlates with introduction of games as examples: overall student feedback remained the same. The average grades improved significantly after introducing mandatory projects based on Lightbot. However, in the following years, exams fell off in quality without changing the contents. Upon introduction of other examples, the average grade improved significantly again.

It may be that breaking the routine of the teaching personnel is more engaging for students. It also is possible that some versions of the projects were

shared between students over years, and parts where copied, resulting in students missing crucial learning outcomes.

Overall, we conclude that teaching – as well as learning – formal methods is hard. Thus, efforts should be taken to improve student engagements. Using games as examples is only one of several possible methodologies.

Acknowledgement. We thank Christoph Heinzen who created several versions of the Pac-Man case study, as well as Philip Höfges for the chess model, AI and GUI.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.
4. M. Baldamus, K. Schneider, M. Wenz, and R. Ziller. Can American Checkers be Solved by Means of Symbolic Model Checking? In *Electronic Notes in Theoretical Computer Science 43*, pages 3–17. Elsevier, 2000.
5. J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour RODIN. 27(8):1065–1084, 2008.
6. M. Carlsson and P. Mildner. SICStus Prolog—the first 25 years. *TPLP*, 12(1-2):35–66, 2012.
7. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proceedings PLILP 1997*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
8. ClearSy. *Atelier B, User and Reference Manuals*, 2016. Available at `http://www.atelierb.eu/`.
9. P. Curzon and P. W. McOwan. Teaching formal methods using magic tricks. In *Fun with Formal Methods: Workshop at the 25th International Conference on Computer Aided Verification*, number 122, 2013.
10. W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović. Verification Games: Making Verification Fun. In *Proceedings FTfJP 2012*, pages 42–49. ACM, 2012.
11. I. Dobrikov, M. Leuschel, and D. Plagge. LTL Model Checking under Fairness in ProB. In *Proceedings SEFM 2016*, pages 204–211. Springer, 2016.
12. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT Solvers for Rodin. In *Proceedings ABZ 2012*, volume 7316 of *LNCS*, pages 194–207. Springer, 2012.
13. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94, Part 2(0):130–143, 2014.
14. M. J. C. Gordon. *HOL: A Proof Generating System for Higher-Order Logic*, pages 73–128. Springer, 1988.
15. J. Hurd. Formal verification of chess endgame databases. Technical report, Oxford University Computing Laboratory, 2005.
16. K. Kennedy, C. Koelbel, and R. Schreiber. Defining and Measuring the Productivity of Programming Languages. *Int. J. High Perform. Comput. Appl.*, 18(4):441–448, Nov. 2004.

17. S. Krings and M. Leuschel. Proof Assisted Symbolic Model Checking for B and Event-B. In *Proceedings ABZ 2016*, volume 9675 of *LNCS*. Springer, 2016.
18. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM 2016*, volume 9681 of *LNCS*. Springer, 2016.
19. P. Körner, J. Bendisposto, J. Dunkelau, S. Krings, and M. Leuschel. Embedding High-Level Formal Specifications into Applications. In *Proceedings FM 2019*, volume 11800 of *LNCS*. Springer-Verlag.
20. L. Ladenberger. *Rapid Creation of Interactive Formal Prototypes for Validating Safety-Critical Systems*. PhD thesis, Heinrich-Heine-Universität Düsseldorf, 2017.
21. L. Ladenberger and M. Leuschel. BMotionWeb: A Tool for Rapid Creation of Formal Prototypes. In *Proceedings SEFM 2016*, volume 9763 of *LNCS*, pages 403–417. Springer, 2016.
22. M. Leuschel. The High Road to Formal Validation:. In *Abstract State Machines, B and Z*, pages 4–23. Springer, 2008.
23. M. Leuschel and J. Bendisposto. Directed Model Checking for B: An Evaluation and New Techniques. In *Proceedings SBMF 2010*, pages 1–16, 2010.
24. M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter 14, pages 427–446. Wiley ISTE, 2014.
25. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME 2003*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
26. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, Feb. 2008.
27. T. Nummenmaa. *Executable Formal Specifications in Game Development*. dissertation, University of Tampere, 2013.
28. T. Nummenmaa, E. Berki, and T. Mikkonen. Exploring Games as Formal Models. In *Proceedings SEEFM 2009*, pages 60–65, Dec 2009.
29. T. Nummenmaa, J. Kuittinen, and J. Holopainen. Simulation As a Game Design Tool. In *Proceedings ACE 2009*, pages 232–239. ACM, 2009.
30. D. Plagge and M. Leuschel. Seven at One Stroke: LTL Model Checking for High-level Specifications in B, Z, CSP, and More. *Int. J. Softw. Tools Technol. Transf.*, 12(1):9–21, Jan. 2010.
31. D. Plagge and M. Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In *Proceedings FM 2012*, LNCS, pages 372–386. Springer, 2012.
32. C. E. Shannon. *Programming a Computer for Playing Chess*, pages 2–13. Springer New York, New York, NY, 1988.
33. E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *Proceedings TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.