Modelling Chess in VDM++

Morten Haahr Kristensen[1] and Peter Gorm Larsen[1]

DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Denmark,
201807664@post.au.dk, pgl@ece.au.dk

Abstract. The game of chess is well-known and widely played all over the world. However, the rules for playing it are rather complex since there are different types of pieces and the ways they are allowed to move depend upon the type of the piece. In this paper we discuss alternative paradigms that can be used for modelling the rule of the chess game using VDM++ and show what we believe is the best model. It is also illustrated how this model can be connected to a standard textual notation for the moves in a chess game. This can be used to combine the formal model to a more convenient interface.

1 Introduction

Many games that humans can play with each other include rules based on logic about what is allowed. Board games often have different kinds of pieces that the players take turn in moving. One of the more complex games is called Chess. In this paper we model the game of chess using VDM++ and discuss the pros and cons of alternative modelling styles [2]. The model was written as an educational example and can be executed to validate whether the rules of a chess game were broken.

The purpose of formal models is to enable formal analysis of desirable properties of the system of interest. However, with each formal model, there are abstraction alternatives and for each of these, it is worthwhile discussing the best paradigms for describing the system in question. In a specification focus is on explaining the key aspects in relation to the purpose of the model and in this context explainability to human beings is much more important than the speed of execution. The explainability of different paradigms will be discussed in this paper using the rules of the game of Chess as an example.

This paper is structured as follows: After this introduction Section 2 provides the reader with a brief introduction to the rules of the Chess game. Afterwards, Section 3 presents reflections about modelling chess using either the object-oriented or the declarative paradigms. Then Section 4 provides the core of the VDM++ model using the declarative paradigm. After that Section 5 briefly explains how the Portable Game Notation can be incorporated enabling one to incorporate a standard textual format as input for the VDM++ model making it easy to take existing games that have been played into the model. Section 6 briefly relates the contribution of this paper with related work. Finally, Section 7 provides a few concluding remarks and considers the future directions.

2 The Rules of Playing Chess

Chess is a two-player turn-based game where one player controls the white pieces and the other controls the black pieces. The game is played on a chessboard that consists of 64 squares (fields) shaped in an 8x8 grid. The columns of the grid are called “files” and the rows are called “ranks”. The squares are coloured alternatively light and dark and a line of squares of the same colour going from one edge to an adjacent edge on the board is called a “diagonal”. The squares following the horizontal axis on the board are annotated using letters the ’a’ – ’h’ and the vertical squares are annotated using the numbers 1 – 8.

Each player initially controls 16 pieces on the board, as seen on fig. 1, where a piece can be discriminated through three different attributes. The first is the piece type, which determines how a piece is allowed to move. The second is the position which determines which square a piece is placed on. Finally, each piece has a colour indicating which player controls it.

Fig. 1. Initial position of the chess pieces where the first row with squares ’a’ – ’e’ shows the position of respectively a rook, knight, bishop, queen and king. The second row shows eight white pawns. Image source: [5].

There are six types of pieces in the game where each player initially controls eight pawns, two rooks, two knights, two bishops, a queen, and, most importantly, a king. Each piece type has a unique way of moving around the board. For example the rook can move along either the file or rank on which it stands and the bishop can only move along the diagonals. A piece can be blocked from moving to a square if there is another piece between the initial and the desired square. However, if an enemy piece occupies the desired square, the player may capture that piece by removing it from the board. When a player can capture a piece, the piece is said to be under attack. The knights have an exception in the moving pattern since they are not blocked by other pieces in their path.

The goal of the game is to put the king of the opposing player in a position where it is impossible to prevent it from being captured in the following turn[1]

There are certain types of moves in chess that can be considered "special" in the regard, that they can only be performed when certain conditions are met. A simple example of such is that a pawn has the option of moving two squares forward from its initial position. Another example is "castling" which is the only type of movement involving two pieces. Castling allows the player to move their king two squares towards a rook on the player's first rank, then move the rook to the square the king just passed. However, castling is only allowed if the king and the rook have not been moved in during the game, if the squares between the king and rook are unoccupied and not under attack, and if the king is not under attack (in check).

If a pawn reaches the rank furthest from its initial position it must change its type to one of the following: Queen, knight, rook or bishop. This type of move is called a "promotion". Taking this into account is actually challenging because it means that the type of that piece is changing dynamically.

3 Alternative Paradigms for Modelling the Game of Chess

The different VDM dialects generally encourage following a Functional Paradigm (FP), but the VDM++ dialect introduces the possibility of using Object-Oriented Paradigm (OOP) for structuring the models. In an OOP setting, there is typically a need to have instance variables inside classes to represent state, and to access and adjust these there is typically a need for operations that need to use the imperative paradigm with assignments to such instance variables. The question of whether or not to use such features arise.

When determining a paradigm to follow, one must consider if it is easier to encapsulate the moving parts or to minimise them. In the case of the game of Chess both OOP and FP paradigms may look appealing as many of the rules of chess generally are stateless and therefore without moving parts. However, the special rules in particular introduce statefulness to the game. This includes example moves such as en passant and castling, where the validity of the moves depends on previous moves.

3.1 Considering the Functional and Object-Oriented Paradigms

Two architectures following OOP and FP were considered. The first follows a typical OOP structure with a base class `Piece` that defines basic methods for determining possible moves. Each piece type then has a subclass implementation defining its unique movement pattern and potential attributes. The special moves would be modelled through `Board`possible_moves` as the `Board` class knows the state of the game. A simplified class diagram of such an architecture can be seen in fig. 2.

The second architecture was written following the FP where only immutable variables were used. In VDM++ this meant defining all data structures as composite types.

[1] For a full list of rules see `https://www.fide.com/FIDE/handbook/LawsOfChess.pdf`.

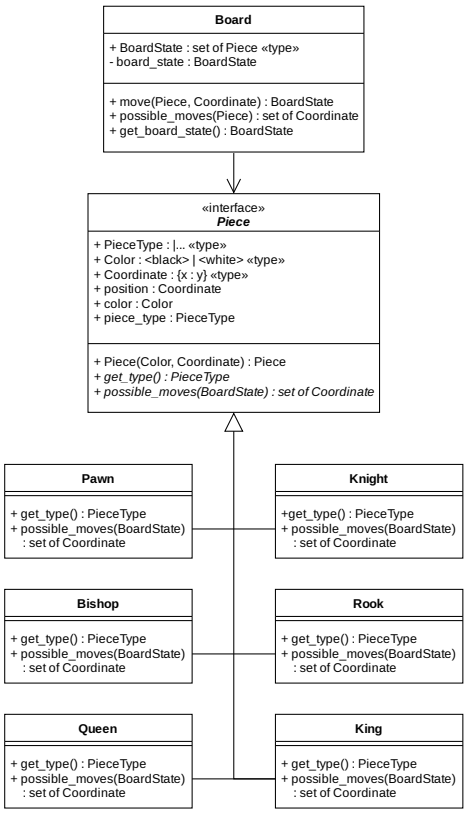

Fig. 2. UML class diagram of the basis for an object-oriented architecture for chess.

The benefit of this is that one only has values in the specification which means that the model is stateless. When writing VDM++ in a functional style, one should consider the classes as modules that encapsulate functionalities together in a namespace (it would thus look more like a VDM-SL model but we have kept it as a VDM++ model in order to ease the comparison). An example of such an architecture can be seen in fig. 3. In this architecture, all pieces share a common data structure containing their colour, position and type.

One benefit of following the FP architecture relates to reasoning about the model. Generally speaking, it can be more difficult to reason about imperative models as their functionality may depend on a global model state. In order to formally verify the behaviour of a function inside an imperative model, one must consider all the methods that can also manipulate the global state. In contrast, a purely functional model consists of referentially transparent functions.

Another benefit relates to testing and verification of the model. When states are introduced to a model the complexity increases significantly as there are more moving parts and thereby more test combinations to be written if the model is to be tested exhaustively. In practice, this means that a model following the FP requires fewer tests to be written as one does not need to consider all the states the model may appear in.

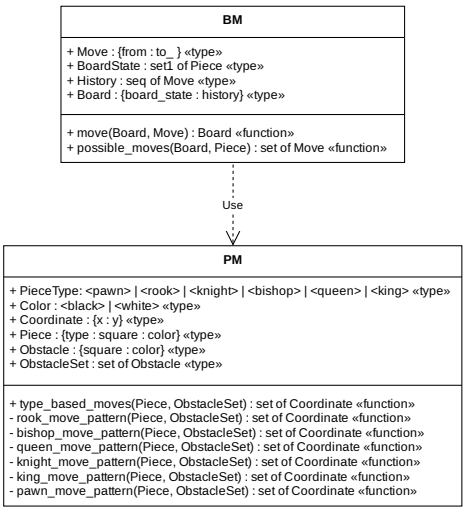

Fig. 3. UML class diagram of the basis for a functional-style architecture for Chess.

However, modelling Chess through an FP architecture also has some limitations, in particular related to the stateful aspects of the game e.g., when modelling castling one needs knowledge of whether the involved king and rook have moved. To determine this with FP one must know the previous moves made in the game and determine if the pieces were involved. With OOP one can simply introduce a boolean attribute on the rooks and kings indicating whether they have moved. While this is also possible following the FP, it implicates all the pieces as inheritance is not available.

At this point, it should be clear that there are benefits to both types of architecture. One should consider whether it is more important to minimise or encapsulate the moving parts when writing the architecture, as this decision may greatly impact the difficulty of the implementation. Finally, one must also consider how exhaustively the model is to be tested as a stateless model should contain less testing combinatorics compared to a stateful model.

An example of how FP can be more elegant than OOP can be found when moving pieces. It is to be assumed that a `Move` is modelled as a composite type of two `Piece`s, one indicating which `Piece` is being moved and the other indicating where it is moved to[2], and that the `BoardState` is a `set of Piece`s. In the OOP case, one would need to find the `Piece` in the current `BoardState` with similar attributes as `Move.from_`, check if a `Piece` is occupying the same square as `Move.to_`, potentially remove that `Piece`, and update the found `Piece` to match the new `Coordinate`. In the case of promotion, one would need to add a new `Piece` of the promoted type and remove the original. When following FP one can simply update the `BoardState` by filtering out the `Piece`s with the `Coordinate`s in the `Move` and making a union with `Move.to_`.

[2] The latter must be a `Piece` over a `Coordinate` to account for promotion.

3.2 Invariants on Compound Types in VDM++

During the early stages of development where the OOP architecture was used, an issue within the VDM tool "VDMJ" [1] lead to an interesting discussion, that at its core relates to the choice of paradigm. The issue relates to the model snippet seen below where a `Piece` is moved on to a given `Coordinate`. The operation updates the state of the `Board` by first removing the captured `Piece` and then updating the position of the moved `Piece`. Since class instances are reference types in VDM++ the latter can be done through the assignment operator directly on the `Piece` as the `board_state` has a reference to the instance. The `board_state` is a `set of Piece` with an invariant stating that the positions of the `Piece`s inside the set must be unique.

```
public move: Piece * Piece`Coordinate ==> ()
move(piece, coord) == (
    let current_coords = state_to_coords_set(board_state) in
        if coord in set current_coords then
            let dead_piece = {p | p in set board_state & p.position = coord} in
                board_state := board_state \ dead_piece;
    piece.position := coord
)
pre piece in set board_state and dead_piece in set board_state;
```

The logic of the model seemed sound but in practice, the interpreter would report an invariant violation when a `Piece` was captured. When debugging the operation it was shown that the invariant was correctly checked when updating the position of the `Piece`, but it was checked against an old `board_state` still containing the `dead_piece`, which caused the invariant to be violated. In short, the issue was caused by an "invariant listener" on the `piece` object that was not correctly updated when the `board_state` was modified[3]. While the exact issue is not a concern of this paper, the complexity of having invariants on compound types containing references is an interesting topic that showcases some of the issues mutability brings.

Since VDM++ objects are references they bring aliasing to the language, i.e., an object like `piece` can be accessed and modified in several places in the specification. If such an object reference is also a member of a compound type instance with an invariant, e.g., `board_state`, the invariant for the compound type must be checked whenever the object is modified. The invariant must also be checked when the compound type instance itself is modified. Furthermore, an object like `piece` could potentially be placed in multiple instances of different compound types with different invariants, which would mean modifying `piece` would result in several invariants being checked. In the case of Chess this might not be an issue as the data structures are relatively flat but in complex industrial cases it may not be the case. The reason for this issue showing up is that in order to make the VDM interpreter efficient while still ensuring that invariants are not violated it only tests the invariants whenever changes are made where the invariants are used. In order for this to work in the presence of aliasing this essentially requires checking the transitive closure of instances connected. This is obviously not efficient so the VDM interpreter ignores such invariants going across instances and thus this is a challenge for the OOP model presented here.

[3] A link to the issue can be found here:
`https://github.com/overturetool/vdm-vscode/issues/197`

Had the specification been written using immutable datatypes like composite types, the invariant issue would not have been a concern, as the `BoardState` would then be a set containing immutable values instead of references. This means that the invariant would only need to be checked when the `board_state` was changed (or following an entirely FP, when a new `BoardState` was created).

While the writer of a specification should not typically concern themselves with the details of the tools, we believe this example makes a strong argument of why following the OOP may introduce unnecessary complexity to a specification. If one was to reason about a specification following the OOP with an instance of a compound type, one would need to consider all the places where the instance was modified but also all the places where the members of the instance could be modified. All of this is without considering more complex OOP concepts such as inheritance that only strengthens the point.

4 Overview of the FP VDM Model of Chess

The model described in this section follows the FP architecture as shown in section 3, where a bottom-up approach will be taken of the most interesting parts of the model.

4.1 PieceModule

The PieceModule (`PM`) class has the responsibility of defining the types necessary to describe a Chess piece and providing the functions necessary to describe their basic movement patterns.

The piece type can be modelled as a union of quote types with the following options: `<pawn>|<rook>|<knight>|<bishop>|<queen>|<king>`. A composite type is used to define the `Coordinate`s which consists of two `nat1` to describe the x- and y-coordinates of a piece. The convention of annotating the x-axis through letters is abstracted away to make the two axes consistent. An invariant is put on `Coordinate` to ensure that the position is legal, i.e., the values are less than nine.

A data structure for the `Piece`s could now be established as a composite type containing the attributes `type : PieceType`, `square : Coordinate`, and `colour : Colour`.

The simple moves, i.e., excluding special moves, can be found for a given `Piece` using the `type_based_moves` function that takes `Piece * ObstacleSet` as parameters and returns a `set of Coordinates` containing the valid `Coordinate`s that the `Piece` can move to. An `ObstacleSet` is needed to filter out the moves that are invalid due to the `Piece`'s path being blocked by an `Obstacle`. An `Obstacle` contains a `Coordinate` and `Colour` where the `Colour` is needed to indicate if the `Obstacle` is capturable or not. Alternatively, one could have modelled the function without the `ObstacleSet` by returning the collection of legal `Coordinate`s assuming the board was empty since this would decouple the `PieceModule` further from the state of the board. However, it would require the `BoardModule` to be more closely coupled to the `PieceType` as it would need a special case for handling the movements of a pawn, as the pawn moves forward but attacks diagonally.

The `type_based_moves` function is written using a cases expression to pattern-match the `PieceType` to a function defining the specific movement pattern of the piece. The movement pattern of the knight and king can be modelled using a common helper function, `possible_move_direction`, as seen in the snippet below. Among the `Piece` and `ObstacleSet`, a pair of ints are provided as parameters to indicate the direction where the possible moves are to be considered. A new `[Coordinate]` is generated based on the `dir` through `coordinate_factory` that returns `nil` if the provided inputs result in an invalid `Coordinate`. `possible_move_direction` evaluates to `nil` if the generated `Coordinate` is invalid or occupied by a friendly piece. Otherwise, the new `Coordinate` is returned.

```
possible_move_direction: Piece * ObstacleSet * (int * int) -> [Coordinate]
possible_move_direction(p, os, dir) ==
let new_c = coordinate_factory(p.square.x + dir.#1, p.square.y + dir.#2)
in
  if (new_c = nil) or
     exists piece in set os &
      (piece.square = new_c) and (piece.colour = p.colour)
  then nil
  else new_c;
```

Having now defined `possible_move_direction` the movement pattern of the knight can be defined as below.

```
knight_move_pattern : Piece * ObstacleSet -> set of Coordinate
knight_move_pattern(p, os) ==
  {possible_move_direction(p, os, mk_(1, 2)), -- 2Up1Right
   possible_move_direction(p, os, mk_(-1, 2)), -- 2Up1Left
   possible_move_direction(p, os, mk_(1, -2)), -- 2Down1Right
   possible_move_direction(p, os, mk_(-1, -2)), -- 2Down1Left
   possible_move_direction(p, os, mk_(2, 1)), -- 1Up2Right
   possible_move_direction(p, os, mk_(-2, 1)), -- 1Up2Left
   possible_move_direction(p, os, mk_(2, -1)), -- 1Down2Right
   possible_move_direction(p, os, mk_(-2, -1)) -- 1Down2Left
   } \ {nil}
pre p.type = <knight>;
```

Similarly, the other piece types rook, bishop and queen has been modelled using the `possible_moves_direction` as seen in the snippet below. This time recursion is needed as these can continue in a direction they are blocked or can capture a piece. Once again a new `[Coordinate]` is generated. The first conditional is identical to the one in `possible_move_direction` but the empty set is returned in this case. A check is then made to determine if an opponent is on the square that is being evaluated, where the recursion is terminated and a set containing the `Coordinate` is returned. At last, the recursive case is defined where a union between the set containing the `Coordinate` and the result of recursively calling the function in the same direction is returned.

```
possible_moves_direction: Piece * ObstacleSet * (int * int) -> set of Coordinate
possible_moves_direction(p, os, dir) ==
  let new_c = coordinate_factory(p.square.x + dir.#1, p.square.y + dir.#2)
  in
     if (new_c = nil) or
        exists piece in set os &
                (piece.square = new_c) and (piece.colour = p.colour)
    then {}
    elseif exists piece in set os &
                (piece.square = new_c) and (piece.colour = opposite_color(p.colour))
```

```
    then {new_c}
    else {new_c} union
           possible_moves_direction(mk_Piece(p.type, new_c, p.colour), os, dir);
```

In principle this recursive function should have a proper `measure` ensuring the termination but it is not straightforward.

The movement pattern of the queen can then be defined as seen below.

```
queen_move_pattern : Piece * ObstacleSet -> set of Coordinate
queen_move_pattern(p, os) ==
  dunion {possible_moves_direction(p, os, mk_(0, 1)),
          possible_moves_direction(p, os, mk_(0, -1)),
          possible_moves_direction(p, os, mk_(1, 0)),
          possible_moves_direction(p, os, mk_(-1, 0)),
          possible_moves_direction(p, os, mk_(1, 1)),
          possible_moves_direction(p, os, mk_(-1, -1)),
          possible_moves_direction(p, os, mk_(-1, 1)),
          possible_moves_direction(p, os, mk_(1, -1))}
pre p.type = <queen>;
```

4.2 BoardModule

The BoardModule (`BM`) class has the responsibility of defining and updating the state of a chessboard. Furthermore, it determines whether or not the special rule moves are possible.

The first type defined in the BM class is the composite type `Move`. Initially, `Move` was modelled as a product type consisting of a `Piece * Coordinate`[4]. This structure made sense up until the point where promotion was implemented since promotion allows for the `PieceType` to be changed, which could not be captured with the old definition. Instead, `Move` was modelled with the attributes `from_` and `to_` that are both of type `Piece`. An invariant was placed upon `Move` that states the following: `m.from_.colour = m.to_.colour and m.from_.square <> m.to_.square` since a `Move` cannot change the colour and must update the position of the `Piece`.

It was then possible to define `History` as a sequence of `Move`s. A sequence was chosen since the ordering of the moves matters and there might be duplicates if a player moves a piece back and forth.

A `BoardState` could then be defined as a `set1 of Piece`, where a set was chosen since the ordering does not matter and duplicates are not allowed as that would indicate two pieces of the same colour and type being placed on the same square. To restrict two `Piece`s from sharing a position the following invariant was written: `forall p1, p2 in set b & p1 <> p2 => p1.square <> p2.square`. Finally, a `Board` type was introduced as a composite type containing a `BoardState` and a `History`.

The function `possible_moves` is responsible for finding the set of valid moves for a `Piece` where both the simple- and special movement patterns are considered. The function finds the set of simple type-based moves, the set of stateful special moves, and the set of illegal stateful moves. It then evaluates to the union between the former two with the set difference of the latter. The logic can essentially be boiled down to “find the

[4] `Move` was changed from a product type to a composite type as the intent is clearer when the fields are named compared to referencing them through ”.#1” and ”.#2”.

entire set of possible moves and remove the impossible ones". Here a set comprehension is used to convert the `Coordinate`s from `simple_moves` to `Move`s through the helper function `piece_coord_to_move`.

```
public possible_moves : Board * PM`Piece -> set of Move
possible_moves (board, piece) == (
  let state_p_moves = stateful_possible_moves(board, piece),
      state_imp_moves = stateful_impossible_moves(board, piece),
      simple_moves = PM`type_based_moves(piece,
          PM`pieces_to_obstacles(board.board_state)) in
          ({piece_coord_to_move(piece, c) | c in set simple_moves} union
              state_p_moves) \ state_imp_moves
)
pre piece in set board.board_state;
```

The function `stateful_impossible_moves` yields the `set of Move` containing `Move`s that result in the player losing, as the rules of Chess disallow such a move from being performed. Thus it contains the moves that put the player in check and if the player already is in check it filters out moves that do not put them out of check. Furthermore, if the `PieceType` is `<pawn>` and promotion is possible then it also contains the `Move` where the pawn moves to a square on the last rank without changing the `PieceType`, as it is illegal to not promote the `Piece`.

`stateful_possible_moves` is a dispatcher that considers the special rules of the pieces, i.e., castling, promotion, en passant and the option for a pawn to move two squares on its first move.

```
stateful_possible_moves: Board * PM`Piece -> set of Move
stateful_possible_moves(board, piece) == (
  cases piece.type:
      <pawn> -> dunion {
          pawn_move_two(board.board_state, piece),
          en_passant(board, piece),
          pawn_promotion(board.board_state, piece)},
      <king> -> castling_possible(board, piece),
      others -> {}
  end
);
```

An example of how a special move can be implemented is seen in the snippet below where promotion is modelled. First, the local definitions `last_y` and `promotable_types` are defined. A set containing the `promotion_squares` is constructed, which is simply the set containing the `Coordinate`s that the pawn could normally move to that are also placed on the last rank. The polymorphic helper function `sets_combine_tuple` is finally used to generate a set of tuples with combinations of `promotable_types` and `promotion_squares`, which can be used to return the set of promotion `Move`s.

```
pawn_promotion: BoardState * PM`Piece -> set of Move
pawn_promotion(board_state, pawn) == (
    let last_y = if pawn.color = <white> then 8 else 1,
        promotable_types = {<knight>, <bishop>, <rook>, <queen>} in
        let promotion_squares = {coord | coord in set
          PM`type_based_moves(pawn, PM`pieces_to_obstacles(board_state))
            & coord.y = last_y} in
            {mk_Move(pawn, mk_PM`Piece(t_c_tuple.#1, t_c_tuple.#2, pawn.color)) |
              t_c_tuple in set sets_combine_tuple[PM`PieceType, PM`Coordinate]
              (promotable_types, promotion_squares)}
)
pre pawn.type = <pawn> and pawn in set board_state;
```

So far the functions have focused on how the valid moves could be determined. Performing a move is done similarly through the function `move` where it is necessary to have different behaviour for castling and en passant as the former changes the `PieceType` and the latter captures a `Piece` without moving to the square. The other types of moves can be modelled through `move_other`. The precondition specifies that the `Move` must be valid and the postcondition specifies that the returned `Board` has a different `BoardState` and a longer `History`.

```
public move: Board * Move -> Board
move(board, mov) ==
  if mov.from_.type = <king> and iss_castling(board, mov)
  then move_castling(board, mov)
  elseif (mov.from_.type = <pawn> and iss_en_passant(board, mov))
  then move_en_passant(board, mov)
  else move_other(board, mov)
pre mov in set possible_moves(board, mov.from_)
  and mov.from_ in set board.board_state
post len board.history < len RESULT.history
  and board.board_state <> RESULT.board_state;
```

The definition of `move_other` is seen below. First, the potentially captured piece is found which is defined in the local definition `dead_piece`. Due to the invariant on `BoardState` it is guaranteed to contain a single `Piece` or be the empty set. The new `BoardState` can then be defined in `new_state` as the previous state without the `dead_piece` and with an updated version of the moved `Piece`. Finally, the new `Board` is returned.

```
move_other: Board * Move -> Board
move_other(board, mov) == (
  let dead_piece = {p | p in set board.board_state & p.square = mov.to_.square} in
      let new_state = (board.board_state \
        (dead_piece union {mov.from_})) union {mov.to_} in
          mk_Board(new_state, [mov] ^ board.history)
)
pre pre_move(board, mov)
post post_move(board, mov, RESULT);
```

Additionally, a helper function `default_board` can be made that defines a board with the initial position as seen in fig. 1 and an empty `History`. This is used as the starting point for new games.

```
public default_board : () -> Board
default_board() ==
(
    let board_state : BoardState = dunion {
      {mk_PM`Piece(<pawn>, mk_PM`Coordinate(x, 2), <white>) | x in set {1,...,8}},
      {mk_PM`Piece(<pawn>, mk_PM`Coordinate(x, 7), <black>) | x in set {1,...,8}}
      -- Repeat for other PieceTypes
    } in
        mk_Board(board_state, [])
);
```

4.3 GameModule

The GameModule (`GM`) class has the responsibility of controlling who has the current turn and declaring the game-winner. The module defines the optional union of quote

types `Winner` with the values `[PM`Color | <remis>]` where `nil` indicates that the game is ongoing. Furthermore, the module defines the composite type `Game` which contains a `Board` and a `PM`Color` indicating the turn.

The function `move` is used to perform a `Move` on the `Board` and potentially determine the winner. First, the `Move` is performed through `BM`move` and saved in the local definition `new_board`. Then it is checked whether the opponent has any valid `Move`s for the next turn. If not then the `Game` is either won by the player or ended in remis, depending on whether the opponent is in check.

```
public move : Game * BM`Move -> (Game * Winner)
move(game, mov) == (
  let new_board = BM`move(game.board, mov),
    opposite_c = PM`opposite_color(game.turn) in
    if forall p in set new_board.board_state &
      p.color = opposite_c => BM`possible_moves(new_board, p) = {} then
      if BM`in_check(new_board.board_state, opposite_c) then
        mk_(game, game.turn)
      else
        mk_(game, <remis>)
    else
      mk_(mk_Game(new_board, opposite_c), nil)
)
pre mov.from.color = game.turn and
  mov in set BM`possible_moves(game.board, mov.from)
post len game.board.history < len RESULT.#1.board.history
    and game.board.board_state <> RESULT.#1.board.board_state;
```

4.4 Runner

The last part of the model consists of a class, `Runner`, that reads the contents of a PGN file (section 5), converts it to `Move`s through the PGN module, and iteratively performs the moves.

From a `Runner` perspective it could also potentially be interesting to create a graphical rendering of the Chess board itself. This could potentially be achieved using either VDMPad [8] or ViennaTalk [9]. In an Overture context this kind of visualisation was also enabled when it was based on Eclipse but this has not yet been fully incorporated in the VSC version [7].

5 Portable Game Notation

The Portable Game Notation [11] (PGN) is the de-facto standard used for chess annotation on many online chess websites.[5] PGN consists of information related to the chess game (e.g. player information) and move text, where the move text describes the actual piece moves of the game. PGN uses letters for the x-axis and numbers for the y-axis as described in section 2. Generally speaking, a move in PGN consists of the `PieceType` as the first character and the `Coordinate` as the second and third characters. Furthermore, there is special notation for castling, check, checkmate, and extra information may be added to remove ambiguity.

[5] PGN is supported on websites such as `chess.com` `https://lichess.org/`, and `chess24.com`.

PGN was added as a way to verify the overall integrity of the VDM model by testing it on some real-world data. Since VDM++ does not include a string manipulation library it was necessary to partly define one. Furthermore, the PGN class has the responsibility of parsing a valid `String` describing a game of chess through PGN notation to the VDM++ `Move` representation and vice versa.

```
values
 numerical_chars = "0123456789";
 numerical_char_to_nat : inmap char to nat =
             {numerical_chars(i) |-> i-1 | i in seq numerical_chars};

 valid_x_chars = "abcdefgh";
 x_char_to_nat1 : inmap char to nat1 =
              {valid_x_chars(i) |-> i | i in seq valid_x_chars};

 piece_type_to_string : inmap PM`PieceType to String =
                        {<pawn> |-> "", <rook> |-> "R", <knight> |-> "N",
                         <bishop> |-> "B", <queen> |-> "Q", <king> |-> "K"}

functions

public move_to_pgn_string: BM`Move -> String
move_to_pgn_string(move) ==
    let piece_type = piece_type_to_string(move.from.type),
        x = (inverse x_char_to_nat1)(move.to_.square.x),
        y = (inverse numerical_char_to_nat)(move.to_.square.y)
     in
        piece_type ^ [x] ^ [y];
```

The helper function `move_to_pgn_string` is used in `Runner` when parsing a chess game and logging the results to a text file. The `PieceType` is mapped to a string by applying the type to `piece_type_to_string`. Similarly, the x- and y-coordinates are found but the inverse mappings are used here, which is possible as the maps are injective.

This class is currently made in VDM but since VDM is not really meant for parsing strings it would make sense to redo this part in Java as a new library for reading this external format directly into VDM structures. It could even use scanner and parser generators inside if desired but this is mainly an issue about the potential error reporting in case the string provided does not live up to the PGN syntax.

6 Related Work

The rules of the chess game have been incorporated in other formalisms as well but from the different publications it is unclear to what extend they have incorporated the more complicated rules such as castling [3,4,10]. Most of the other formal methods analysis of the chess game are considering this from a model checking perspective. Thus, the focus is primarily in analysing the potential future outcome of a game of chess or simply looking into whether a sequence of moves are legal or not. In contrast the work presented in this paper focus more on the most natural way to represent the rules of the chess game. However, in [4] there is an attempt of first generalising the rules of games before specialising it to different games such as chess. It is possible that this approach could be reused more systematically in the work presented in this paper with advantage as well.

7 Concluding Remarks and Future Work

This article presented a feature-complete model of Chess in VDM++ that can be used as an educational example or as a basis for formal analysis of other topics related to the game of Chess. The model has been added as one of the models that can be imported into the Visual Studio Code version of Overture [6] so others can also experiment with the entire VDM++ model.

In general, we find that there are interesting pros and cons when considering which paradigm to use for modelling a system in VDM++. In the case of the game of Chess, we find that the functional paradigm works better than the object-oriented paradigm (OOP), but it also has drawbacks. If OOP is chosen for a given system, one must be aware of the complications it may bring, in particular relating to invariants across object references. The same challenge would also appear for operations with post-conditions because this also could relate to instance variables in other objects before execution of the operation.

Acknowledgements We would like to thank the anonymous reviewers for valuable feedback on the original version of this paper.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005). https://doi.org/10.1007/b138800, http://overturetool.org/publications/books/vdoos/
3. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: Integrating formal specifications into applications: the ProB Java API. Formal Methods in System Design **58**(1), 160–187 (2021)
4. Krings, S., Körner, P.: Prototyping Games Using Formal Methods. In: Cerone, A., Roggenbach, M. (eds.) Formal Methods – Fun for Everybody. pp. 124–142. Springer International Publishing, Cham (2021)
5. Lichess: Chess board editor, https://lichess.org/editor
6. Lund, J., Jensen, L.B., Macedo, H.D., Larsen, P.G.: Towards UML and VDM Support in the VS Code Environment. In: Macedo, H.D., Pierce, K. (eds.) Proceedings of the 20th International Overture Workshop. pp. 51–66. Overture (7 2022)
7. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
8. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormaliSE 2015. pp. 33–39. In connection with ICSE 2015, Florence (May 2015)
9. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)

10. Saralaya, V., Kishore, J.K., Reddy, S., Pai, R.M., Singh, S.: Modeling and Verification of Chess Game Using NuSMV. In: Abraham, A., Lloret Mauri, J., Buford, J.F., Suzuki, J., Thampi, S.M. (eds.) Advances in Computing and Communications. pp. 460–470. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Wikipedia: Portable Game Notation (2007), https://en.wikipedia.org/w/index.php?title=Portable_Game_Notation&oldid=1101954154