

Winning Strategies in Quoridor

An analysis of the strategy game Quoridor

J.E.P.M. van Laarhoven

Department of Mathematics and Computer Science

Eindhoven University of Technology

Eindhoven, The Netherlands

j.e.p.m.v.laarhoven@student.tue.nl

Abstract—The game of Quoridor is a two or four player strategy game that is based on simple concepts, but due to the complexity of the rules has a large state space that approximately has $6.5595 \cdot 10^{44}$ possible states. Using model checking, small instances of the game can be analyzed. In this paper we present that Quoridor has been strongly solved for smaller instances of the game: The starting player has a winning strategy on a square board of even size and the opponent has a winning strategy on square boards of odd size. Using the mCRL2 toolset and language, Labelled Transition Systems (LTSs) have been generated for the solved instances of the games, showing the winning strategy for the winning player.

Index Terms—Quoridor, Strategy Games, Formal Verification, mCRL2, Model Checking, Strongly Solved

1. INTRODUCTION

During the past decades lots of researchers have shown interest in the field of solving strategy games like Checkers [1], Connect Four [2–4] and Awari [5]. Especially in the field of Artificial Intelligence (AI) a lot of interest is shown for solving such games or developing agents to beat a human. In recent years, model checking tools like mCRL2 [6], NuXMV [7, 8], SPIN [9] and UPPAAL [10] have been introduced and optimized. Using these tools, one is able to create a finite-state model and verify whether it meets a given specification or set of properties. Hence, these model checking tools are suitable for solving such strategy games. In this report, we show the effectiveness of such a model checking tool when solving the strategy game Quoridor.

Quoridor is a two or four player strategy game designed by Mirko Marchesi that was introduced in 1997 by Gigamic Games [11]. The game is played on a board consisting of 9 rows of 9 squares each, with grooves in between the squares that allow fences to be placed between the squares. At the start of the game, the pawns of the players are positioned on opposite sides of the board and the number of fences are equally divided among the players. The aim of the game is to reach the opposite side of the board. The first player to reach their opposing side wins the game.

Throughout the game, the players take turns in which they may move their pawn to an adjacent square, jump over their opponent if they are located next to them, or place a fence which could interfere with the path of the opponent, as players are not allowed to move through/over the placed fences. Throughout the game, the goal for each player should remain accessible from their current position.

The game is based on rather simple concepts, but certain more specific rules make the game more complex, which results in a high state space complexity of approximately $6.5595 \cdot 10^{44}$ states. We provide more details on this in section 2.4. Generally, a higher state space complexity means that the model checker needs to do more calculations, and therefore it requires a larger time and space complexity.

In this study we have used the mCRL2 language [6] and toolset [12] to model and verify whether winning strategies exist in the two-player variant of Quoridor. The mCRL2 language [6] is a formal specification language that extends the algebra of communicating processes (ACP [13]). To check whether certain properties hold in a model specification, one can express the properties using the first-order modal μ -calculus. The mCRL2 toolset contains a rich set of tools that enable the automatic transformation from a mCRL2 model specification into a Linear Process Specification (LPS), Labelled Transition System (LTS) and, together with a property expressed using the first-order modal μ -calculus, a Parametrized Boolean Equation System (PBES). In addition to these transformation tools, the mCRL2 toolset also contains tools that optimize and analyze these LPSs, LTSs and PBESs.

Using the mCRL2 language and toolset we created a mCRL2 model for Quoridor. We have found that for smaller instances of the game there always is a winning strategy for one of the players of the game. The results suggest a pattern that there is a winning strategy for player 2 on odd sized boards and for player 1 on even sized boards. Using the tool, we generated Labelled Transition Systems (LTSs) that show the winning strategy for these players. These LTSs show for each step of the opponent, what the next step for the player is according to the winning strategy.

Outline: In section 2 we discuss the notion of “solved” and discuss which and how games have been solved over the years, as well as a discussion on how Quoridor relates to other strategy games and the research that has been performed on this game. In section 3, we introduce the game of quoridor in which we provide some terms and definitions to formally model and analyse the game. Furthermore, we informally discuss the complete set of rules of Quoridor. In section 4, we discuss the mCRL2 model in detail and prove the correctness of the model. In section 5, we discuss the results we obtained when analyzing the Quoridor game and present the winning strategies as a set of guidelines. Finally, in section 6 we

discuss future research that could be performed to optimize the model and to formulate the winning strategies as a number of guidelines on how to play the game.

2. RELATED WORK

In this section we discuss the research that has been performed on solving strategic games. We discuss how a selection of games has been solved, what research has been performed on Quoridor and how Quoridor relates to the other strategy games.

To compare Quoridor to other strategic games, we first have to define the definition of “solved”. Many notions of “solved” are used to specify whether strategies have been found for one or both players in these games. We use the more fine-grained notion of “solved” as presented by L.V. Allis in [14]. Allis distinguishes between three types of solutions for a game: *ultra-weakly solved*, *weakly solved* and *strongly solved* games.

A game is said to be *ultra-weakly solved* if a proof exists which proves that the outcome of a game is known from the initial position. Games are *weakly solved*, if the outcome of the game is known from the starting position, and a strategy is known that guarantees this outcome. A game is said to be *strongly solved* if for any position in the game, the most optimal move can be determined within reasonable time. Table I contains examples of games that have been ultra-weakly, weakly and strongly solved.

Notion of “Solved”	Game	Research
Ultra-Weakly Solved	Hex	[15]
	Checkers	[1]
Weakly Solved	Domineering	[16–18]
	Fanorona	[19]
	Go-Moku	[14]
	Nine Men’s Morris	[20]
	Qubic	[21]
	Renju	[22]
Strongly Solved	Awari	[5]
	Connect Four	[2, 3, 23]
	Kalah	[24]
	Tic-Tac-Toe	[25]

TABLE I: Examples of games that have been solved

In sections 2.1 to 2.3 we discuss three two-player strategy games that have been solved over the years: Connect Four, Checkers and Awari. In section 2.4 we discuss how Quoridor compares to other strategy games and what research has been performed on Quoridor.

2.1 Connect Four

Connect Four is a well-known two player strategy board game, in which the players, white and black, take turns inserting colored markers into a vertical grid of 7×6 cells, aiming to create a horizontally, vertically or diagonal line of four markers of their color.

Connect Four has been weakly solved independently by J.D. Allen [26] and L.V. Allis [27] in 1988. In “The Complete Book of Connect 4”, Allen presents the winning strategies that he found when analyzing the game. Allis used a knowledge-based approach with nine strategy rules. He implemented a Shannon C-type strategy program called VICTOR based on these strategy rules. Using this program, Allis found that on a board with 6 or 7 columns and an even number of rows, a winning strategy for white exists, the starting player, if he opens the game by placing his first marker in the middle column. Furthermore, if white does not place his first marker in the middle column, then black has a strategy such that the game ends in a draw.

In 1995, J. Tromp strongly solved the game using a database of all 8-ply positions and their theoretical results [23]. Using this database, the most optimal move from any position in the game could be evaluated. Tromp implemented a benchmark called “Fhourstones” that he used to weakly solve the outcomes of games on boards of x rows and y columns, for integers x and y , where $x + y \leq 15 \wedge x, y \geq 4$ [28].

2.2 Checkers

Checkers, also called 8×8 draughts, is another widely known two player strategy game that is played on an 8×8 checkerboard. Both player black and white start with 12 stones each. The first player that captures all the stones of his opponent, wins the game.

In 2001, M. Baldamus et al. [29] ultra-weakly solved checkers for boards of size 3×3 , 4×4 and 5×5 . Using the symbolic model checker SymQuest, Baldamus et al. verified that there is a winning strategy for white, on a board of 3×3 . Furthermore, they showed that a game on a board of size 4×4 would lead to a draw, and that on a board of 5×5 a winning strategy exists for black.

In 1989, Schaeffer et al. started implementing a computer program called Chinook that is able to compute the most optimal move for a player in each turn. In 2007, Schaeffer et al. published the article “Checkers Is Solved” [1], in which they show that the game of checkers has been weakly solved, i.e., from the starting positions the game always result in a draw if both players play perfectly.

2.3 Awari

Awari is a 3500 year old game that originates from Africa. The game is played on a board with 12 pits, 6 per player, plus 1 auxiliary pit per player. At the end of the game, the player with the most stones in their auxiliary pit wins the game. J.W. Romein and H.E. Bal strongly solved Awari [5] using a database to calculate the most optimal move from any position in the game. From this database, they concluded that if both players play optimally, the game ends in a draw.

2.4 Quoridor

In this section we discuss the state space complexity of Quoridor, which is used as a metric to compare the complexity of Quoridor to other strategic board games. Furthermore, we discuss the research that has been performed on Quoridor.

2.4.1 State Space Size

We use S for the upperbound of the state space complexity of a two-player game of Quoridor, the state space complexity of Quoridor can be calculated using equation (1), where $\#_p$ is the number of players, S_p is the number of ways in which the pawns can be located on the board and S_f is the number of ways in which (a subset of) the fences can be placed on the board. We include the number of players as a factor of the state space, since in each possible state of the game, it can be any player's turn.

$$S = \#_p \cdot S_p \cdot S_f \quad (1)$$

We introduce N for the number of squares on one row/column of the Quoridor board and F for the number of fences that are assigned to each player. We define $S^{(N,F)}$ as the state space complexity of a game on a board with $N \times N$ squares in which each player initially has F fences. Similarly, for S_p and S_f we introduce $S_p^{(N,F)}$ and $S_f^{(N,F)}$. The state space complexity can be calculated for any combination of N and F using equation (2).

$$S^{(N,F)} = \#_p \cdot S_p^{(N,F)} \cdot S_f^{(N,F)} \quad (2)$$

The pawn of the first player can be positioned on any of the $N \times N$ squares. As each pawn must be positioned on a different square, it follows that the pawn of the second player can only be positioned on $(N \times N) - 1$ squares. Hence, the state space complexity considering only the positions of the fences can be calculated using equation (3).

$$S_p^{(N,F)} = N^2 \cdot (N^2 - 1) \quad (3)$$

According to the rules of the game, each fence must be placed between two pairs of squares. Hence, there are $N - 1$ of such pairs on each row and column of the board. Therefore we have $(N - 1)^2$ possible positions at which the first fence can be placed on the board both horizontally and vertically, thus in total $2 \cdot (N - 1)^2$ possible positions.

When a fence is placed on the board, the number of positions at which we can place the next fence is decreased by 2, 3 or 4. Each fence can block 4 positions for other fences, which includes the location at which the fence is placed. However, two placed fences may be blocking the same position for a fence. The number of possible placements is decreased by at least 2, as the location at which the fence has been placed cannot contain another fence in neither direction, as fences may not overlap.

For our comparison we assume that placing a fence reduces the number of positions for the next fence by 2. This results in a larger state space complexity compared to 3 and 4, and therefore leads to an overestimation. Using this approach, we also neglect the rule that the goal line must remain accessible for both players. Therefore, the state space complexity would be lower than the value computed, thus overestimating the state space complexity. Hence, the state space complexity computed in equation (4) is an overestimation of the actual complexity.

$$S_f^{(N,F)} = \sum_{i=0}^{2 \cdot F} \left(\prod_{j=0}^{i-1} ((2 \cdot (N-1)^2) - (2 \cdot j)) \right) \quad (4)$$

We can estimate the total state space complexity of a game on a board of size $N \times N$, in which initially each player has F fences, using equations (2), (3) and (4), the full formula is shown in equation (5).

$$S^{(N,F)} = \#_p \cdot N^2 \cdot (N^2 - 1) \cdot \sum_{i=0}^{2 \cdot F} \left(\prod_{j=0}^{i-1} ((2 \cdot (N-1)^2) - (2 \cdot j)) \right) \quad (5)$$

In table II the estimations of the state space complexity are given for all $3 \leq N \leq 9$ and $0 \leq F \leq 10$. Quoridor is normally played on a board with 9 by 9 squares, where both players get 10 fences each ($N = 9$, $F = 10$). According to equation (5), the state space complexity is estimated as $6.5595 \cdot 10^{44}$.

In table III, a list of two-player strategy games is presented showing the size of the state space, as log to base 10, and to which extend the game has been solved.

		F										
		0	1	2	3	4	5	6	7	8	9	10
N	3	2	3	4	4	4	4	4	4	4	4	4
	4	2	5	7	9	10	11	11	11	11	11	11
	5	3	6	8	11	14	16	18	20	21	21	21
	6	3	6	10	13	16	19	22	25	27	30	32
	7	3	7	11	14	18	21	25	28	31	34	37
	8	3	7	11	15	19	23	27	30	34	38	41
	9	4	8	12	16	20	24	28	32	36	40	44

TABLE II: State Space Complexity (as log to base 10) for each game instance, a color scale is used from blue, the smallest state space complexity, to red, the largest state space complexity.

Game	$\log_{10}(\text{SSC})$	Solved
Tic Tac Toe	3	Strongly Solved
Nine Men's Morris	10	Weakly Solved
Awari	12	Strongly Solved
Kalah	13	Strongly Solved
Connect Four	13	Strongly Solved
Domineering	15	Weakly Solved
Checkers	18/20	Weakly Solved
Fanorona	21	Weakly Solved
Qubic	30	Weakly Solved
Quoridor	44	-
Chess	44	Partially Strongly Solved
Hex	57	Partially Ultra-Weakly Solved
Go-Moku	105	Partially Weakly Solved

TABLE III: State Space Complexity (SSC) of, a selection of, Strategy Games that have been (Partially) Solved [30]

According to this table, we see that Quoridor has a higher State Space complexity than any of the strongly or weakly solved games, and equal (as log to base 10) to chess which only has been strongly solved for games with a 3 to 7-piece end game, or smaller instances of the board.

Various games have been solved using the mCRL2 language and toolkit, examples can be found in the mCRL2 GitHub repository [31]. For example, Connect Four has been strongly solved for all boards of size (c, r) where $1 \leq x \leq 7$ and $1 \leq y \leq 6$, excluding $(7, 5)$, $(6, 6)$ and $(7, 6)$. A game of Connect Four on a board of 4 rows and 7 columns leads to a model with a state space of 6.741.832.166, or as log to base 10, 9.

This suggest that using mCRL2, we should be able to solve all games of Quoridor in which each player gets up to 1 fence each, as well as all games on a 3×3 and 4×4 boards and games with 2 fences per player for boards up to 6×6 .

2.4.2 Research on Quoridor

Quoridor was introduced in 1997 and is therefore, compared to other games like Connect Four, Checkers and Awari, still a rather young game. Therefore, the game has not received a lot of attention yet. Only a handful of research papers were published about Quoridor.

Most attempts to solve Quoridor use artificial intelligence or reinforcement learning. Although the number of research papers that have been published on this topic is limited, most researchers tried solving the game using an AI agent with either the MiniMax [32–36] or NegaMax [32] decision strategy. Respal et al. [35] and Jose et al. [36] both tried to solve Quoridor using reinforcement learning, using Genetic and Monte Carlo Tree Search algorithms. Unfortunately, none of these researchers were able to develop a Quoridor agent that is able to beat a human.

3. QUORIDOR

In this section, we go into more detail about the game Quoridor. In sections 3.1 and 3.2 we introduce the notation and definitions used in the remainder of this paper. In section 3.3 we informally present the complete set of rules of Quoridor.

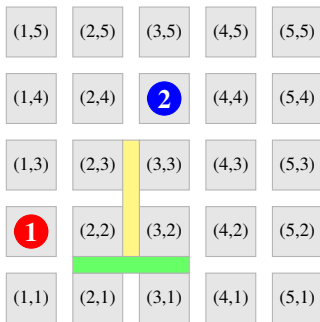


Fig. 1: Example of Board Notation

3.1 Notation

In this section we introduce a number of terms that are used in the remainder of the paper.

3.1.1 Board

A game of quoridor is played on a board of squares, separated by grooves. We define N as the number of squares that are on 1 row/column of the board. A *board* is defined as a grid of N by N squares, separated by grooves. We only consider boards of N by N squares where $3 \leq N \leq 9$. The game is originally played on a board of 9 by 9 squares, that is, $N = 9$.

3.1.2 Square (Location)

We introduce a coordinate system to uniquely identify a square on the board. Each square is identified by a coordinate pair: (column, row).

In Figure 1, an example board is shown where $N = 5$. For all unoccupied squares their corresponding coordinate pair is shown. Player 1 (red) is located at position $(1, 2)$ and player 2 (blue) is located at position $(3, 4)$.

3.1.3 Fence

A fence is a small wooden block that is placed within the grooves between the squares. The length of each fence is equal to the length of 2 squares plus the groove that separates these squares. We introduce F , $0 \leq F \leq 10$, as the number of fences that each player gets at the start of the game. Thus, the game is played with $2 \cdot F$ fences in total. The original game is played with 20 fences, 10 per player ($F = 10$).

3.1.4 Touching

We say that a fence f is *touching* a square s if, and only if, f occupies any of the 4 grooves that are adjacent to s .

For example, in Figure 2, we have that a fence f would touch square $(2, 2)$ if, and only if, a part of f is located in any of the yellow marked grooves. In Figure 1, we have that the yellow fence is touching squares $(2, 2)$, $(2, 3)$, $(3, 2)$ and $(3, 3)$.

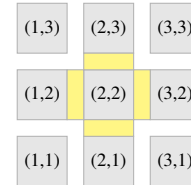


Fig. 2: Example Touching: A fence is touching square $(2, 2)$ if, and only if, a part of the fence is located in any of the yellow marked grooves.

3.1.5 Fence Location

We express the location of a fence based on the two pairs of squares that it touches and a direction: H (horizontally) or V (vertically). Let T be the set of all squares that some fence f

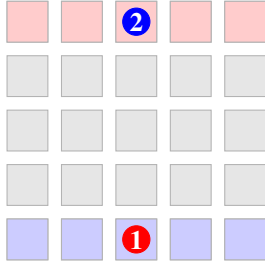


Fig. 3: The layout of the board at the start of the game.

touches, the location of a fence f is expressed by the following triple:

$$\left(\min_{t \in T} \text{col}(t), \min_{t \in T} \text{row}(t), d \right),$$

where $\text{col}(t)$, $\text{row}(t)$ and $\text{dir}(t)$ denote the column, row and direction of a coordinate triple respectively.

In Figure 1, the green and yellow fences are placed at locations $(2, 1, H)$ and $(2, 2, V)$ respectively.

3.1.6 Base & Goal Line

The players start in the middle column of opposing rows on the board. The row on which a player starts is called his *base line*. The aim of the game is to reach a square in the base line of the opponent. For each player, the base line of the opponent is referred to as his *goal line*.

In Figure 3 an example board of 5×5 squares is shown, in which the base and goal line of player 1 consist of all red and blue squares respectively.

3.2 Definitions

In this section we define a set of terms that are used in the remainder of this paper.

Definition 1 (Adjacent). Let s and s' be two squares on the board, s and s' are adjacent if and only if it holds that

$$|\text{col}(s) - \text{col}(s')| + |\text{row}(s) - \text{row}(s')| = 1$$

For example, in Figure 1, we have that square $(3, 3)$ is adjacent to squares $(2, 3)$, $(3, 2)$, $(3, 4)$ and $(4, 3)$.

Definition 2 (Blocking). Let s and s' be two adjacent squares on the board and let L_f denote the list of locations of all fences that are placed on the board. A move from square s to adjacent square s' is blocked if, and only if:

- s and s' are in the same row, i.e., $\text{row}(s) = \text{row}(s')$, and one of the following two conditions holds:
 - $(\min(\text{col}(s), \text{col}(s')), \text{row}(s), V) \in L_f$
 - $(\min(\text{col}(s), \text{col}(s')), \text{row}(s) - 1, V) \in L_f$
- s and s' are in the same column, i.e., $\text{col}(s) = \text{col}(s')$, and one of the following two conditions holds:
 - $(\text{col}(s), \min(\text{row}(s), \text{row}(s')), H) \in L_f$
 - $(\text{col}(s) - 1, \min(\text{row}(s), \text{row}(s')), H) \in L_f$

For example, in Figure 1, we have that the move from square $(3, 2)$ to both $(2, 2)$ and $(3, 1)$ are blocked. The moves to $(3, 3)$ and $(4, 2)$ are not blocked.

Definition 3 (Overlap). Let f_1 and f_2 be two fences placed at fence positions fp_1 and fp_2 . Fences f_1 and f_2 overlap if, and only if, one of the following conditions hold:

- They have the same location, that is: $fp_1 \approx fp_2$
- They are crossing each other, that is:

$$\text{col}(fp_1) \approx \text{col}(fp_2) \wedge \text{row}(fp_1) \approx \text{row}(fp_2)$$

$$\wedge \text{dir}(fp_1) \not\approx \text{dir}(fp_2)$$

- They share a groove between one pair of vertically adjacent squares, that is:

$$\text{dir}(fp_1) \approx \text{dir}(fp_2) \approx H \wedge |\text{col}(fp_1) - \text{col}(fp_2)| = 1$$

$$\wedge \text{row}(fp_1) \approx \text{row}(fp_2)$$

- They share a groove between one pair of horizontally adjacent squares, that is:

$$\text{dir}(fp_1) \approx \text{dir}(fp_2) \approx V \wedge |\text{row}(fp_1) - \text{row}(fp_2)| = 1$$

$$\wedge \text{col}(f) \approx \text{col}(fp_2)$$

Definition 4 (Behind). Let s and s' be adjacent squares. If s and s' are adjacent and are in the same row, then we say that square $(\text{col}(s') - (\text{col}(s) - \text{col}(s')), \text{row}(s))$ is behind s' from the perspective of s . Similarly, if s and s' are adjacent and in the same column, square $(\text{col}(s), \text{row}(s') - (\text{row}(s) - \text{row}(s')))$ is behind s' from the perspective of s .

For example, in Figure 1, we have that square $(5, 2)$ is behind square $(4, 2)$ from the perspective of square $(3, 2)$.

Definition 5 (Access). Player p has access to all locations s for which hold that there exist a sequence of non-blocked moves between adjacent squares from the current location of p to s .

3.3 Rules

In section 1 we briefly discussed the Quoridor game and a selection of its rules. In this section, the rules are presented in more detail. Note that these rules are still expressed informally, the rules are formalised in section 4.5. These rules are based on the official rules by Mirko Marchesi [11].

3.3.1 Rules related to the start of the game

- R1: Each player starts with F fences.
- R2: At the start of the game, each player places their pawn in the middle of opposing sides of the board. That is, players 1 and 2 place their pawns at $(\lfloor \frac{N}{2} \rfloor + 1, 1)$ and $(\lfloor \frac{N}{2} \rfloor + 1, N)$ respectively.
- R3: Player 1 starts first.
- R4: The players play in turn.
- R5: A player's turn consist of either moving their pawn or placing one of his fences on the board.

3.3.2 Rules related to moving a pawn

- R6: A player may move his pawn from location s to s' , if:
- a) there is no a fence in between s and s' ;
 - b) s' is not occupied by the opponent;
 - c) the move is allowed by rule R7, R8 or R9.

R7: A player may move his pawn to any adjacent square, either horizontally or vertically.

In Figure 4, the possible moves to adjacent squares are shown as covered in rules R6 and R7. In this example, the player is only allowed to move to the square above and to the right of his current square. The other squares are blocked by fences.

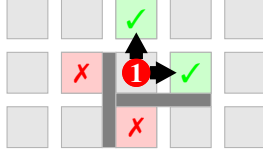


Fig. 4: Example of Rule 6 & 7

R8: If the player is in a square that is adjacent to the square of the opponent, then he is allowed to jump over the opponent.

In Figure 5, the possible moves to adjacent squares are shown as covered in rules R6 and R7. Since the opponent is in the right adjacent square, the player may not move to this square by rule R6b. However, the player is allowed to jump over the player by rule R8.

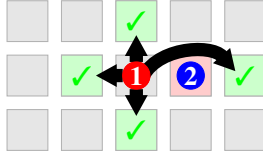


Fig. 5: Example of Rule 6 & 8

R9: A player may make a blocked jump from current location s to s'' , when the opponent is positioned at location s' , if the following conditions hold:

- Both s and s'' are adjacent to s' .
- There is a fence blocking the move from s' to the position behind s' , from the perspective of s .

In Figure 6, the possible moves to adjacent squares are shown as covered in rules R6 and R7. In this example, the player may not jump over the opponent as discussed in rule R8, as this violates rule R6A due to the fence placement. However, the player is allowed to make a blocked jump to the square above the opponent by rule R9.

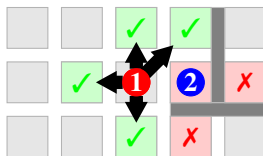


Fig. 6: Example of Rule 6 & 9

3.3.3 Rules related to placing a fence

- R10: A player may only place a fence between two pairs of squares.
- R11: Fences may not overlap.
- R12: When placing a fence, access to the goal line for all players must be maintained.

3.3.4 Rules related to the end of the game

- R13: After a player reaches any square of his goal line, he wins the game.
- R14: Once a player wins the game, the game ends.

4. MODELLING QUORIDOR

In this section, we discuss the mCRL2 model that has been designed and implemented to model Quoridor and to find winning strategies. We present the sorts (section 4.1), the functions (section 4.2), the external actions (section 4.3) and the process (section 4.4) of the Quoridor model, which altogether model the game and its behavior. In section 4.5, we formalize the rules from section 3.3 using the actions of the model. In section 4.6, we prove that the behavior of the Quoridor model satisfies the rules as discussed in section 3.3. A complete version of the model can be found in Appendix A.

4.1 Sorts

We have designed and implemented 4 sorts (types) that are used to model the game. These sorts are shown in Figure 7, in mCRL2 notation.

```

1 sort
2   Position = struct pos(col:Pos, row:Pos);
3   FPosition = struct fpos(col:Pos, row:Pos, dir:Direction);
4   Direction = struct H | V;
5   Turn = struct P1 | P2 | None;

```

Fig. 7: Sorts implemented in the Quoridor Model

Here `Pos` is the predefined sort of mCRL2 consisting of all positive numbers.

4.1.1 Sort *Position*

The `Position` type expresses a coordinate pair. As mentioned in section 3.1, a square on the board is uniquely identified by a coordinate pair (column, row). In the mCRL2 model, these are expressed as an object of type `Position`. For example, square (3,4) would be expressed as `pos(3,4)` in the model.

For any instance `p` of type `Position` we have projection functions `col(p)` and `row(p)`, which respectively give the column and row of `p`. For example, if `p ≈ pos(3,4)`, then `col(p) ≈ 3` and `row(p) ≈ 4`.

We write $p_1 \approx p_2$ for `Position` instances p_1 and p_2 if, and only if, the column and row of p_1 and p_2 are equal. We write that the column and row of `Position` instances p_1 and p_2 are equal by $p_1 \approx_c p_2$ and $p_1 \approx_r p_2$ respectively.

4.1.2 Sort FPosition

The `FPosition` type expresses a fence location as a triple. As mentioned in section 3.1, the location of a fence on the board is uniquely identified by a triple (column, row, direction). In the `mCRL2` model, these are expressed as an object of type `FPosition`. E.g., a fence at location (3,4,H) would be expressed as `fpos(3,4,H)` in the model.

For any instance `fp` of type `FPosition` we have projection functions `col(fp)`, `row(fp)` and `dir(fp)` that respectively give the row, column and direction of the coordinate triple. For example, if `fp ≈ fpos(3,4,H)`, then `col(fp) ≈ 3`, `row(fp) ≈ 4` and `dir(fp) = H`.

We write `fp1 ≈ fp2` for `FPosition` instances `fp1` and `fp2` if, and only if, the column, row and direction of `fp1` and `fp2` are equal. We write that the column, row and direction of `FPosition` instances `fp1` and `fp2` are equal by `fp1 ≈c fp2`, `fp1 ≈r fp2` and `fp1 ≈d fp2` respectively.

4.1.3 Sort Direction

The `Direction` type expresses a the direction of a fence. As mentioned in section 3.1, the direction of a fence is either horizontal or vertical. In the `mCRL2` model, these are respectively expressed as `H` and `V`.

4.1.4 Sort Turn

The `Turn` type expresses which player performed or may perform an action. In this paper, we only consider games of two players. Hence, in the `mCRL2` model, each instance of type `Turn` can be expressed as `P1`, `P2` or `None`. An instance of type `Turn` can only have the value `None` after the game ends, that is, when one of the players reach their goal line.

4.2 Functions

In this section, we discuss the functions and mappings that are used within the `Quoridor` model. For each of the functions we discuss its goal, the input, output and the preconditions.

4.2.1 N and F

Functions `N` and `F` are used as constants in the model. `N` expresses the number of squares on one row/column of the board, `F` expresses the number of fences that each player owns at the start of a game. Hence, the total number of fences that could be placed on the board is $2 \times F$.

4.2.2 Function upperMiddle

Function `upperMiddle` calculates the upper middle number of a range of numbers. The function has no input variables, instead the function uses the value of `N`. The function outputs the upper middle number in the sequence $1..N$. The output is of type `Pos`. The `upperMiddle` function is used to determine the column in which both players are positioned at the start of the game. Algorithm 1 describes the implementation of the `upperMiddle` function using pseudocode.

Algorithm 1 upperMiddle

```
1: return  $\lfloor N/2 \rfloor + 1$ 
```

4.2.3 Function validPos

Function `validPos` determines whether a given position `pA` of type `Position` is valid, that is, whether `pA` is located on the board. On a board with `N` columns/rows, we have that position `pA` is valid if, and only if, both its column and row are within the range $[1..N]$. Algorithm 2 describes the implementation of the `validPos` function using pseudocode.

Algorithm 2 validPos(pA)

```
1: return  $1 \leq \text{col}(pA) \leq N \wedge 1 \leq \text{row}(pA) \leq N$ 
```

4.2.4 Function isBlocked

Function `isBlocked` determines whether a move from one square to an adjacent square is blocked by a fence. The function requires the following inputs:

- `fences`: The list of all the locations of fences that are currently placed on the board, which is of type `FSet(FPosition)`;
- `pA`: The position at which the player's pawn is currently located, which is of type `Position`;
- `pD`: The position to which the player's pawn is potentially moved, which is of type `Position`.

The output is of type `Bool`, `true` is returned if the path from `pA` to `pD` is blocked by a fence, `false` otherwise. Note that, this function requires that `pA` and `pD` are adjacent squares. Algorithm 3 describes the implementation of the `upperMiddle` function using pseudocode.

Algorithm 3 isBlocked(pA, pD, fences)

▷ Squares `pA` and `pD` must be adjacent.

```
1: if not validPos(pD):
2:   return false
3: if pA ≈r pD:
4:   minCol ← min(col(pA), col(pD))
5:   if fpos(minCol, row(pA), V) ∈ fences:
6:     return true
7:   if fpos(minCol, row(pA)-1, V) ∈ fences:
8:     return true
9: else
10:  minRow ← min(row(pA), row(pD))
11:  if fpos(col(pA), minRow, H) ∈ fences:
12:    return true
13:  if fpos(col(pA)-1, minRow, H) ∈ fences:
14:    return true
15: return false
```

In line 1 of algorithm 3, we check if the destination position is valid. If this position is not valid, it means that a player wants to move their pawn off the board, which is trivially not allowed, hence we directly return `false`. In line 3, we check whether `pA` and `pD` are in the same row. If this is the case, then there are 2 fence positions which would block the jump, as illustrated by the blue and red fence position in the right board of Figure 8. In lines 11 and 13 we check whether there

is a fence in these positions respectively. Similarly, if the pA and pD are in the same column, there are 2 fence positions that could block the jump, as illustrated in the left board of Figure 8, which are checked in lines 5 and 7.

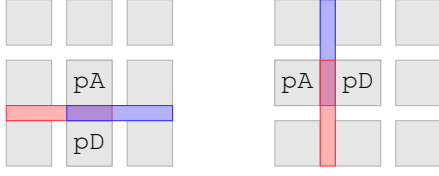


Fig. 8: Fence positions that block the move between positions pA and pD .

4.2.5 Function *isValidMove*

Function *isValidMove* determines whether moving a pawn from one location to another location is valid. The function requires the following inputs:

- pA : The position at which the player's pawn is currently located, which is of type *Position*;
- pB : The position at which the opponent's pawn is currently located, which is of type *Position*;
- pD : The position to which the player's pawn is potentially moved, which is of type *Position*;
- *fences*: The list of all the locations of fences that are currently placed on the board, which is of type *FSet* (*FPosition*).

The function returns an instance of type *Bool*, *true* if the move is valid, *false* otherwise. Rules 6 to 9 of section 3.3 specify which moves are considered to be valid.

We distinguish 3 types of moves: (1) a move to an adjacent square, (2) a jump over the opponent to a square in the same row or column and (3) a jump over the opponent to a square that is in different row and column. We use the Manhattan distance to calculate the distance between two positions.

To determine whether the move from position pA to position pD is valid, we have to check whether a set of conditions holds. Algorithm 4 describes the implementation of the *isValidMove* function using pseudocode.

4.2.6 Function *isValidPlace*

Function *isValidPlace* determines whether placing a fence on the board at fence-position fp of type *FPosition* is valid by calling multiple sub-functions. The function requires the following inputs:

- pA : The position at which the pawn of player 1 is located, which is of type *Position*;
- pB : The position at which the pawn of player 2 is located, which is of type *Position*;
- fp : The fence-position at which either of the player possibly places a fence, which is of type *FPosition*;
- *fences*: The list of all the locations of fences that are currently placed on the board, which is of type *FSet* (*FPosition*).

Algorithm 4 *isValidMove* ($pA, pB, pD, fences$)

```

1:  $rd \leftarrow |col(pA) - col(pD)| + |row(pA) - row(pD)|$ 
2: if  $rd > 2 \vee rd = 0 \vee pB \approx pD \vee \text{not } \text{validPos}(pD)$ :
3:   return false
4: if  $rd = 1$ : ▷ Move of type (1)
5:   return not isBlocked( $pA, pD, fences$ )
6: else if  $rd = 2 \wedge (pA \approx_c pD \vee pA \approx_r pD)$  ▷ Type (2)
7:   if not isBlocked( $pA, pB, fences$ ):
8:     if not isBlocked( $pB, pD, fences$ ):
9:       return  $pB$  is adjacent to both  $pA$  and  $pD$ 
10: else if  $rd = 2 \wedge pA \not\approx_c pD \wedge pA \not\approx_r pD$  ▷ Type (3)
11:    $pC \leftarrow$  position behind  $pB$  from the perspective of  $pA$ .
12:   if not isBlocked( $pA, pB, fences$ ):
13:     if not isBlocked( $pB, pD, fences$ ):
14:       if isBlocked( $pB, pC, fences$ ):
15:         return  $pB$  is adjacent to both  $pA$  and  $pD$ 
16: return false

```

Algorithm 5 *isValidPlace* ($pA, pB, fp, fences$)

```

▷ We abbreviate isGoalReachable to iGR.
1:  $t \leftarrow \text{countFenceTouching}(fp, fences)$ 
2: if  $t \geq 2 \vee (t = 1 \wedge fp \text{ is touching the border})$ :
3:   if iGR( $P1, [pA], \{\}, \{fp\} + fences$ ):
4:     if iGR( $P2, [pB], \{\}, \{fp\} + fences$ ):
5:       return true
6: return false

```

The output of the function is of type *Bool* and is *true* if, and only if, the fence does not overlap with any of the already placed fences, and the fence does not block the access to the goal line for any of the players.

Note that this would be an expensive computation if we did not add additional checks to improve the efficiency. We want to limit the number of *isGoalReachable* calls. Hence, we first check whether the fence that we are trying to add touches any of the already placed fences. We do not have to check if the goal lines remain accessible for both players if the fence that is placed only touches 1 of the already placed fences and does not touch the border of the board. Assuming that before the fence was placed both players had access to their goal lines, then after the fence is placed, there still exists one or more paths for both players to reach their finish line. As such a fence placement always allows the players to shift there path around this fence. Algorithm 5 describes the implementation of the *isValidPlace* function using pseudocode.

4.2.7 Function *addBlocked*

Function *addBlocked* is used to generate the set of fence locations that are blocked when a fence is placed at location fp of type *FPosition*. The function requires the following inputs:

- *blocked*: The list of all the fence-locations that are either occupied by a fence on the board, or fence-locations for which hold that if a fence would be placed

on that position, it would overlap with an already placed fence. This input is of type `FSet (FPosition)`.

- `fp`: The location at which a fence is placed on the board, which is of type `FPosition`.

The output of the function is of type `FSet (FPosition)`, which contains all of the fence locations that were in the given input set `blocked`, plus all fence locations that would overlap with fence location `fp`. Algorithm 6 describes the implementation of the `addBlocked` function using pseudocode.

By keeping track of such locations in the model, we prevent that the `isValidPlace` function is called for these locations, which gives a small performance gain as function `isValidPlace` could be expensive computation.

Algorithm 6 `addBlocked(fp, blocked)`

▷ We use the notation $a \stackrel{\cup}{\leftarrow} b$ to write that $a \leftarrow a \cup b$

```

1: blocked  $\stackrel{\cup}{\leftarrow}$  {fp}
2: if dir(fp)  $\approx$  H:
3:   blocked  $\stackrel{\cup}{\leftarrow}$  {fpos(col(fp), row(fp), V)}
4:   blocked  $\stackrel{\cup}{\leftarrow}$  {fpos(col(fp) - 1, row(fp), H)}
5:   blocked  $\stackrel{\cup}{\leftarrow}$  {fpos(col(fp) + 1, row(fp), H)}
6: else
7:   blocked  $\stackrel{\cup}{\leftarrow}$  {fpos(col(fp), row(fp), H)}
8:   blocked  $\stackrel{\cup}{\leftarrow}$  {fpos(col(fp), row(fp) - 1, V)}
9:   blocked  $\stackrel{\cup}{\leftarrow}$  {fpos(col(fp), row(fp) + 1, V)}
```

4.2.8 Function `isGoalReachable`

Function `isGoalReachable` determines if there exists a path for one of the players from their current location to any square on their goal line. The function requires the following inputs:

- `p`: The turn instance of the game, specifying for which player we check whether their goal line is still reachable, which is of type `Turn`;
- `lv`: The list of squares that have been discovered to be reachable from the player's location, but have not yet been processed. The input `lv` is of type `List (Position)`;
- `lp`: The list of squares that have been discovered to be reachable from the player's location, and have been processed. The input `lp` is of type `List (Position)`;
- `fences`: The list of all the locations of fences that are currently placed on the board, which is of type `FSet (FPosition)`.

The function uses the Depth-First Search (DFS) algorithm to determine whether a path from the player's current position to any square on their goal line exist. Algorithm 7 describes the implementation of the `isGoalReachable` function using pseudocode.

The output of the function is of type `Bool`, which is `true` if, and only if, there exists a path from player `p`'s current location to any square on his goal line.

Algorithm 7 `isGoalReachable(p, lv, lp, fences)`

▷ Checks whether `p` has a path to his goal line.
 ▷ Initially `lv` is a list with 1 element: the player's position.
 ▷ Initially `lp` is an empty list.

```

1: while lv  $\neq$  []
2:   pA  $\leftarrow$  pop-head(lv)
3:   if row(pA)  $\approx$  if (p  $\approx$  P1, N, 1) return true
4:   lp  $\leftarrow$  lp + {pA}
5:   for all valid positions pB
6:     if pA and pB are adjacent
7:       if not isBlocked(pA, pB, fences)
8:         if pB  $\notin$  lv  $\wedge$  pB  $\notin$  lp
9:           lv  $\leftarrow$  lv + {pB}
10: return false
```

4.2.9 Function `countFenceTouching`

Function `countFenceTouching` calculates the number of fences that would be touched when a fence is placed at fence-location `fp` of type `FPosition`. The function requires the following inputs:

- `fp`: The fence-location for which we count how many other fences it touches when we place a fence at that location, which is of type `FPosition`;
- `fences`: The list of all the locations of fences that are currently placed on the board, which is of type `FSet (FPosition)`.

For each fence location `fp`, there are up to 10 fence locations that would be touching `fp` if a fence would be located at that position. These 10 locations are shown in Figure 9, in which the left and right column show the 10 location for a horizontal and vertical fence location respectively. The function returns the number of fences that would be touched, which is of type `Nat`. Algorithm 8 describes the implementation of the `countFenceTouching` function using pseudocode.

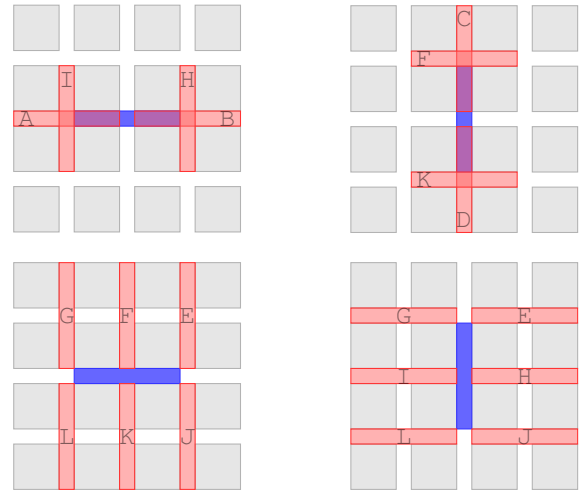


Fig. 9: For both a horizontal (left column) and vertical (right column) placed fence, the fence locations that touch this blue fence are indicated using red fences.

Algorithm 8 `countFenceTouching(fp, fences)`

▷ We use the notation $a \stackrel{\cup}{\leftarrow} b$ to write that $a \leftarrow a \cup b$

```
1:  $t \leftarrow \{\}$ 
2:  $dInv \leftarrow \text{if}(\text{dir}(fp) \approx H, V, H)$ 
3: if  $\text{dir}(fp) \approx H$ :
4:    $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) - 2, \text{row}(fp), H)\}$ 
5:    $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) + 2, \text{row}(fp), H)\}$ 
6: else
7:    $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp), \text{row}(fp) - 2, V)\}$ 
8:    $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp), \text{row}(fp) + 2, V)\}$ 
9:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) + 1, \text{row}(fp) + 1, dInv)\}$ 
10:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp), \text{row}(fp) + 1, dInv)\}$ 
11:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) - 1, \text{row}(fp) + 1, dInv)\}$ 
12:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) + 1, \text{row}(fp), dInv)\}$ 
13:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) - 1, \text{row}(fp), dInv)\}$ 
14:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) + 1, \text{row}(fp) - 1, dInv)\}$ 
15:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp), \text{row}(fp) - 1, dInv)\}$ 
16:  $t \stackrel{\cup}{\leftarrow} \{\text{fpos}(\text{col}(fp) - 1, \text{row}(fp) - 1, dInv)\}$ 
17: return  $\text{size}(t \cap \text{fences})$ 
```

▷ Fence A (Only for horizontal placed fences)
▷ Fence B (Only for horizontal placed fences)
▷ Fence C (Only for vertical placed fences)
▷ Fence D (Only for vertical placed fences)
▷ Fence E
▷ Fence F
▷ Fence G
▷ Fence H
▷ Fence I
▷ Fence J
▷ Fence K
▷ Fence L

4.3 Actions

The Quoridor model consists of three actions: `movePawn`, `addFence` and `win`. In this section, the description of each action is given, as well as the required arguments.

4.3.1 `movePawn(p, c, r)`

Action `movePawn(p, c, r)` denotes the event that player p move his pawn to position (c, r) . Note that, p should either be $P1$ or $P2$, as no moves are allowed when the game has ended and $p \approx \text{None}$ denotes a game that has ended. Arguments c and r represent the column and row to which the player moves his pawn, they should both be positive integers in the range $[1..N]$. Arguments p is of type `Turn`.

4.3.2 `addFence(p, c, r, d)`

Action `addFence(p, c, r, d)` denotes the event that player p places a fence on fence-position (c, r, d) . Note that, p should either be $P1$ or $P2$, as no moves are allowed when the game has ended and $p \approx \text{None}$ denotes a game that has ended. Arguments c and r should both be positive integers in the range $[1..N]$. Arguments p and d are of type `Turn` and `Direction` respectively.

4.3.3 `win(p)`

Action `win(p)` denotes the event that player p won the game, that is, reached his goal line. Note that, p should either be $P1$ or $P2$. After `win(P1)` or `win(P2)` occurs, no other actions can be performed, that is, we reach a deadlock. Arguments p is of type `Turn`.

4.4 Game Process

The Game process models the behavior of a game of Quoridor. The process keeps track of the following data:

- `turn`: The player whose turn it currently is, which is of type `Turn`;
- `p1` and `p2`: The current locations of player 1 and 2 respectively, which both are of type `Position`;
- `rfl`: The number of fences that are assigned to player 1 and have not been placed on the board yet, this number is of type `Nat`;
- `fences`: The list of all the locations of fences that are currently placed on the board, which is of type `FSet(FPosition)`;
- `blocked`: The list of all the fence-locations that are blocked as there is already a fence placed at that location, or because placing a fence at that location would result in overlapping fences. The `blocked` parameter is of type `FSet(FPosition)`.

The process has been modelled such that only the player that corresponds to the current `turn` value can perform a step. The model only allows the player to move his pawn to a location that is valid, or to add a fence to a valid location on the board if the player still has fences to place on the board. If at any point, one of the players reaches their goal line, then directly after the last action `movePawn(turn, c, r)` occurs, a `win(turn)` action occurs. If this action occurs, we reach a deadlock, indicating that the game has ended.

After the `win(p)` action occurs, the parameters of the process are reset to default values, which reduces the state space of the model.

Note that we only keep track of the number of fences that are assigned to the first player that he did not place yet. This is because the remaining number of fences that the opponent still can place on the board can be calculated using F , the number of fence locations in `fences` and `rfl`.

4.5 Formalization of the Rules

In section 3.3 the rules of Quoridor were informally presented. In this section, we use the actions as discussed in section 4.3 to formalize the rules of Quoridor. We define Fx as the formalisation of rule Rx . We use an underscore ($_$) to denote that an argument of one of the actions is not relevant, and is allowed to be replaced with any value of the type of that argument.

4.5.1 Rules related to the start of the game

- F1: Actions $\text{addFence}(P1, _, _, _)$ and $\text{addFence}(P2, _, _, _)$ shall at most occur F times each.
- F2: At the start of the game, the pawns of players 1 and 2 are positioned at $(\lfloor \frac{N}{2} \rfloor + 1, 1)$ and $(\lfloor \frac{N}{2} \rfloor + 1, N)$ respectively.
- F3: No $\text{movePawn}(P2, _, _)$ or $\text{addFence}(P2, _, _, _)$ action may occur before either a $\text{movePawn}(P1, _, _)$ or $\text{addFence}(P1, _, _, _)$ action occurs.
- F4: For all $p, q \in \{P1, P2\}$ of type Turn , where $p \not\approx q$, it must hold that after a $\text{movePawn}(p, _, _)$ or $\text{addFence}(p, _, _, _)$ action occurs, no $\text{movePawn}(p, _, _)$ or $\text{addFence}(p, _, _, _)$ action may occur before either a $\text{movePawn}(q, _, _)$ or $\text{addFence}(q, _, _, _)$ action occurs.
- F5: Let rf1 be the number of fences that are assigned to player 1 that he has not yet placed on the board. Furthermore, let p denote the current turn of the game. Then, there must exist at least one action $\text{movePawn}(p, _, _, _)$ that can be fired and if player p still has remaining fence to place and there exists a FPosition fp at which a fence can be placed, then there must at least be one action $\text{addFence}(p, \text{col}(\text{fp}), \text{row}(\text{fp}), \text{dir}(\text{fp}))$ that can be fired.

4.5.2 Rules related to moving a pawn

- F6: a) After an action $\text{addFence}(_, c, r, d)$ occurs, for some c, r in range $[1..N]$, then for all $p \in \{P1, P2\}$ of type Turn , if action $\text{movePawn}(p, \text{col}(\alpha), \text{row}(\alpha))$ happens, then action $\text{movePawn}(p, \text{col}(\beta), \text{row}(\beta))$ may not happen before an action $\text{movePawn}(p, c', r')$ has occurred, where $\text{pos}(c', r') \not\approx \text{pos}(\text{col}(\beta), \text{row}(\beta))$, for all $(d, \alpha, \beta) \in$

```
{
  (H, pos(c, r),      pos(c, r+1)),
  (H, pos(c+1, r),    pos(c+1, r+1)),
  (H, pos(c, r+1),    pos(c, r)),
  (H, pos(c+1, r+1),  pos(c+1, r)),
```

```
(V, pos(c, r),      pos(c+1, r)),
(V, pos(c+1, r),    pos(c, r)),
(V, pos(c, r+1),    pos(c+1, r+1)),
(V, pos(c+1, r+1),  pos(c, r+1))
}
```

- b) For all $p, q \in \{P1, P2\}$ of type Turn , where $p \not\approx q$, if $\text{movePawn}(p, c, r)$ is the last movePawn action that occurred for player p , then $\text{movePawn}(q, c, r)$ may not occur until another $\text{movePawn}(p, _, _)$ action occurs.

F789: We combine rules R7 until R9 into one formalized rule, which also directly incorporates rule R6: If after the last $\text{movePawn}(p, c, r)$ action another $\text{movePawn}(p, c', r')$ action follows, then it must hold that $\text{isValidMove}(\text{pos}(c, r), pB, \text{pos}(c', r'), \text{fences})$ returns true , where $p \in \{P1, P2\}$ and pB is the location of the opponent.

4.5.3 Rules related to placing a fence

- F10: Action $\text{addFence}(_, c, r, _)$ may only occur for c, r in range $[0..(N-1)]$.
- F11: Let $d_1, d_2 \in \{H, V\}$ and $d_1 \not\approx d_2$, then after $\text{addFence}(_, c, r, d_1)$ occurs, the following actions may no longer occur:

```
addFence(\_, c, r, d1),
addFence(\_, c, r, d2),
addFence(\_, c - if(d1 ≈ H, 1, 0),
          r - if(d1 ≈ V, 1, 0), d1)
addFence(\_, c + if(d1 ≈ H, 1, 0),
          r + if(d1 ≈ V, 1, 0), d1)
```

- F12: After an $\text{addFence}(_, c, r, d)$ action occurs, it must hold that both $\text{isGoalReachable}(P1, [\text{pA}], \{\}, \{\text{fpos}(c, r, d)\} + \text{fences})$ and $\text{isGoalReachable}(P2, [\text{pB}], \{\}, \{\text{fpos}(c, r, d)\} + \text{fences})$ return true , where pA and pB are the current positions of players 1 and 2 respectively and fences is the set of all locations of fences that are placed on the board.

4.5.4 Rules related to the end of the game

- F13: After a $\text{movePawn}(P1, _, N)$ action, a $\text{win}(P1)$ action occurs. Similarly, after a $\text{movePawn}(P2, _, 1)$ action occurs a $\text{win}(P2)$ action occurs.
- F14: After a $\text{win}(_)$ action occurs, we reach a deadlock.

4.6 Correctness of the Quoridor Model

We can verify the behavioral correctness of the model by translating the formalized rules from section 4.5 into modal μ -calculus formulae. The way in which these rules are formulated allows for a trivial conversion to modal μ -calculus formulae. For each formalized rule, except for rules F2 and F10, the modal μ -calculus formulae can be found in Appendix B. We did not translate formalized rules F2 and F10 as these are more easily proven on the structure of the model.

4.6.1 Rule F2

According to rule F2, players 1 and 2 must be positioned at squares $(\lfloor \frac{N}{2} \rfloor + 1, 1)$ and $(\lfloor \frac{N}{2} \rfloor + 1, N)$ respectively at the start of the game. Figure 10 shows the initial state of the Quoridor model. In the initial state we have a process `Game`, which keeps track of the state of the game. The second and third argument of this process keep track of the position of player 1 and 2 respectively. Hence, since in our initial states these positions are set to `pos(upperMiddle + 1, 1)` and `pos(upperMiddle + 1, N)`, and `upperMiddle` is defined as $\lfloor \frac{N}{2} \rfloor$, therefore rule F2 trivially holds.

```

1  init
2  allow ({
3      movePawn, addFence, win
4  },
5      Game(
6          % Player 1 starts the game
7          P1,
8
9          % Start position of player 1
10         pos(upperMiddle + 1, 1),
11
12         % Start position of player 2
13         pos(upperMiddle + 1, N),
14
15         % Number of fences per player, the
16         % set of fences placed on the board
17         % and the set of fence locations that
18         % are blocked by other fences.
19         F, {}, {}
20     )
21 );

```

Fig. 10: Initial State of the Quoridor Model

4.6.2 Rule F10

As shown in the `Game` process in Appendix A, an `addFence(t, c, r, d)` action can only occur for some $t : \text{Turn}, c, r : \text{Pos}, d : \text{Direction}$ if, among others, it holds that both `pos(c, r)` and `pos(c+1, r+1)` are valid. Thus, it must hold that $1 \leq c, r \leq N$ and $1 \leq c+1, r+1 \leq N$, which simplifies to $1 \leq c, r \leq N-1$. Hence, it should always follow that the fences are placed between two pairs of squares, this proves that rule F10 holds.

4.6.3 Rule F13

As discussed in section 4.5, rule F13 trivially holds. There are no actions that allow for removal of fences, hence a fence can never be removed from the board once it has been placed.

4.6.4 Correctness

Using the mCRL2 toolset we have verified that, the behavior of, the Quoridor model satisfies the formalized rules of the Quoridor game. Using the toolset, we have verified that the behavior of the Quoridor model satisfies all the modal μ -calculus formulae. We have proven this for all instances of the game for which the winning strategies have been determined, as discussed in section 5.3. As the translation of the formalized rules into modal μ -calculus formulae is straightforward, we can therefore assume that the modal μ -calculus formulae are correct with regard to the formalized rules. Therefore, since all formulae are satisfied in the Quoridor model, we may conclude that the behavior of the model is in line with the formalized rules.

5. WINNING STRATEGIES

In this section we discuss how the modal μ -calculus can be used to denote that a player has a winning strategy for some instance, a combination of N and F , of the game. Using the tools of the mCRL2 toolset, we have verified whether these modal μ -calculus formulae are satisfied for some instance of the game. We discuss the analysis procedure used to obtain these results and the results themselves.

5.1 Modal μ -Calculus Formula

To check whether properties hold on a model, the mCRL2 toolset uses the first-order modal μ -calculus. Using a modal μ -calculus formula, we can express that player 1 or 2 has a winning strategy for an instance of the game.

Before we can express such a formulae, we first need to define what is considered as a winning strategy. In Quoridor, a player has a winning strategy if, and only if, the player has a sequence of actions from the initial state of the game that leads to a win for that player, no matter which actions are performed by the opposing player.

More formally, we say that a player p has a winning strategy from a state s if, and only if, one of the following holds in s :

- It is player p 's turn and there exist at least one action that player p can perform which lead to a state in which p has a winning strategy;
- It is opposing player's turn and all actions that this player can perform lead to states in which p has a winning strategy.
- Player p is located on his goal line.

Hence, player p has a winning strategy if he has a winning strategy from the initial state.

We can express this property using the least fixed point operator of the modal μ -calculus. Since we use a least fixed point, we only consider games with a finite number of steps. The modal μ -calculus formula describing that player 1 has a winning strategy is shown in Figure 11.

Similarly, we can express that player 2 has a winning property using a modal μ -calculus formula, which is shown in Figure 12.

```

1 mu X.(
2   % Player 1 wins
3   <win(P1)>true ||
4
5   % Or, there exists a move for player 1 such that:
6   <exists c,r:Pos.(movePawn(P1, c, r) || exists d:Direction.addFence(P1, c, r, d))>
7   (
8     % 1. He directly wins the game
9     <win(P1)>true ||
10
11    % Or, 2. For all moves possible moves of player 2, player 1 can force a win
12    (
13      forall c2,r2:Pos.forall d2:Direction.[movePawn(P2, c2, r2) || addFence(P2, c2, r2, d2)]X
14    )
15  )
16 )

```

Fig. 11: Player 1 has a winning strategy

```

1 % For all possible opening moves by player 1
2 forall c,r:Pos.forall d:Direction.[movePawn(P1, c, r) || addFence(P1, c, r, d)]
3 (
4   mu X.
5   % Player 2 wins
6   <win(P2)>true ||
7
8   % Or, there exists a move for player 2 such that:
9   <exists c2,r2:Pos.exists d2:Direction.movePawn(P2, c2, r2) || addFence(P2, c2, r2, d2)>
10  (
11    % 1. He directly wins the game
12    <win(P2)>true ||
13
14    % Or, 2. For all moves possible moves of player 1, player 2 can force a win
15    (
16      forall c3,r3:Pos.forall d3:Direction.[movePawn(P1, c3, r3) || addFence(P1, c3, r3, d3)]X
17    )
18  )
19 )

```

Fig. 12: Player 2 has a winning strategy

5.2 Analysis

In order to verify whether there is a winning strategy for either player, we first convert the mCRL2 model into a linear process specification using the mcrl22lps tool. On this LPS we apply a set of tools to reduce the size and complexity of the LPS, these are lpssuminst, lsparunfold, lpsrewr and lpsconstelm. After which we convert the LPS to a LTS using the lps2lts tool. We reduce this LPS modulo strong bisimilarity using the ltsconvert, which reduces the state space by a small factor. From the reduced LTS, we generate a PBES for each of the modal μ -calculus formulae expressing the winning conditions using the tool lts2pbes. Finally, we solve the tool using pbessolve. The complete set of commands, including arguments, to run the analysis is shown in Figure 13.

We have ran our analysis on the Mastodont server at Eindhoven University of Technology. The Mastodont is designed to support long-running computations that require a large amount of RAM. The server runs Ubuntu Server 20.04 LTS and is equipped with 4 Intel(R) Xeon(R) Gold 6136 CPU's with a clockspeed of 3.00GHz and 3 terabytes of RAM.

```

1 mcrl22lps -v Quoridor-N-F.mcRL2 Quoridor-N-F.lps
2
3 lpssuminst -v -sBool Quoridor-N-F.lps
4   | lsparunfold -v -l -n5 -sPosition
5   | lpsrewr -v
6   | lpsconstelm -v -c
7   | lpsrewr -v Quoridor-N-F-1.lps
8
9 lps2lts --cached -v -rjittyc --threads=16
10   Quoridor-N-F-1.lps Quoridor-N-F.lts
11
12 ltsconvert -v -ebisim Quoridor-N-F.lts
13   Quoridor-N-F-1.lts
14
15 % The command below will be run for both players ,
16 % where  $\phi$  should be replaced by the name of
17 % the modal  $\mu$ -calculus formulae which expresses
18 % that the player has a winning strategy.
19 lts2pbes -v -c -p -f" $\phi$ " Quoridor-N-F-1.lts
20   | pbessolve -v --threads=16 -rjittyc -s1
21   --file=Quoridor-N-F-1.lts
22   --evidence-file=Quoridor-N-F- $\phi$ -evidence.lts

```

Fig. 13: Overview of Analysis

		F										
		0	1	2	3	4	5	6	7	8	9	10
N	3	2	2	2	2	2	2	2	2	2	2	2
	4	1	1	1	1	1	1	1	1	1	1	1
	5	2	2	2	?	?	?	?	?	?	?	?
	6	1	1	?	?	?	?	?	?	?	?	?
	7	2	2	?	?	?	?	?	?	?	?	?
	8	1	1	?	?	?	?	?	?	?	?	?
	9	2	2	?	?	?	?	?	?	?	?	?

?

X

1

2

B

Not verified

Neither player has a winning strategy

Only player 1 has a winning strategy

Only player 2 has a winning strategy

Both players have a winning strategy

Fig. 14: Winning Strategies in Quoridor for each Instance of the Game

5.3 Results

Using the mCRL2 toolset, we have verified whether the modal μ -calculus formulae from section 5.1 hold for a number of instances on the game, that is, a combination of board size N and number of fences per player F. Figure 14 presents the results of the verified instances. The number inside the square denotes which player has a winning strategy. Game instances that have not been verified are indicated with a question mark.

As shown in Figure 14, we have only been able to solve games for which the state space complexity, as log to base 10, is smaller than or equal to 8. As the Mastodont server is a shared server, there was not sufficient memory available to analyze any instances of the game with a higher state space complexity than 8.

For all instances of the game that have been analyzed, we have generated labelled transition systems that consist of a counter example in case the modal μ -calculus formula was not satisfied in the model and contains the evidence graph

in case the formula was satisfied in the model. For example, in Figure 15, the evidence graph is shown for the modal μ -calculus formula expressing that player 2 has a winning strategy on a board with 3×3 squares, where each player has 0 fences. In this graph, the green and red marked states denote the initial state and the state in which player 2 has won the game respectively. The number in the states denotes which player's turn it is in that state. In the evidence graph we combined all winning states to improve the readability. In Figure 16, a graphical representation of the evidence graph of Figure 15 is shown using board notation. In this figure, the red and blue arrow denote a move of players 1 and 2 respectively. The green arrows indicate the $\text{win}(P2)$ action, these are not included in the evidence graph.

The mCRL2 model, all modal μ -calculus formulae and all evidence graphs for all instances of the game that have been analyzed can be requested from the author.

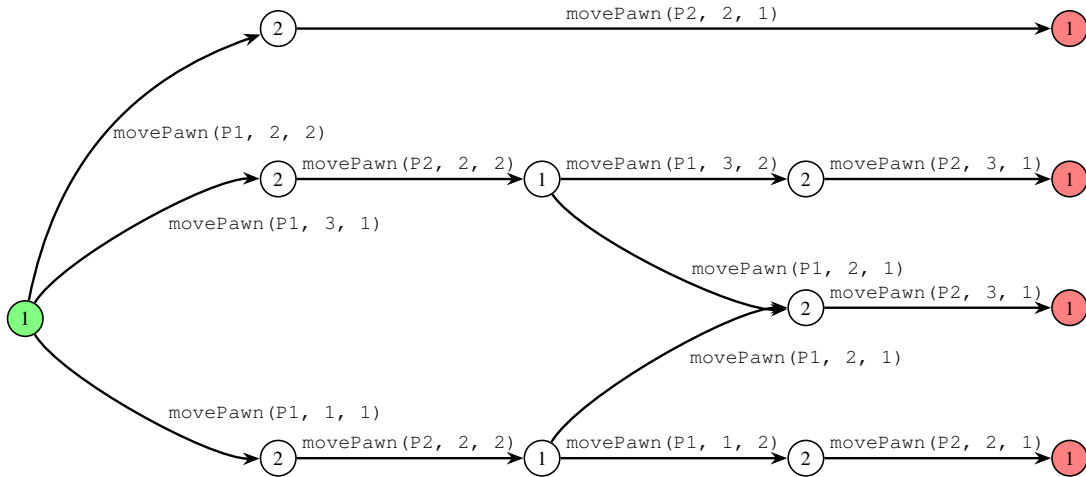


Fig. 15: Evidence Graph: Player 2 has a Winning Strategy on a board of 3×3 squares and 0 fences per player.

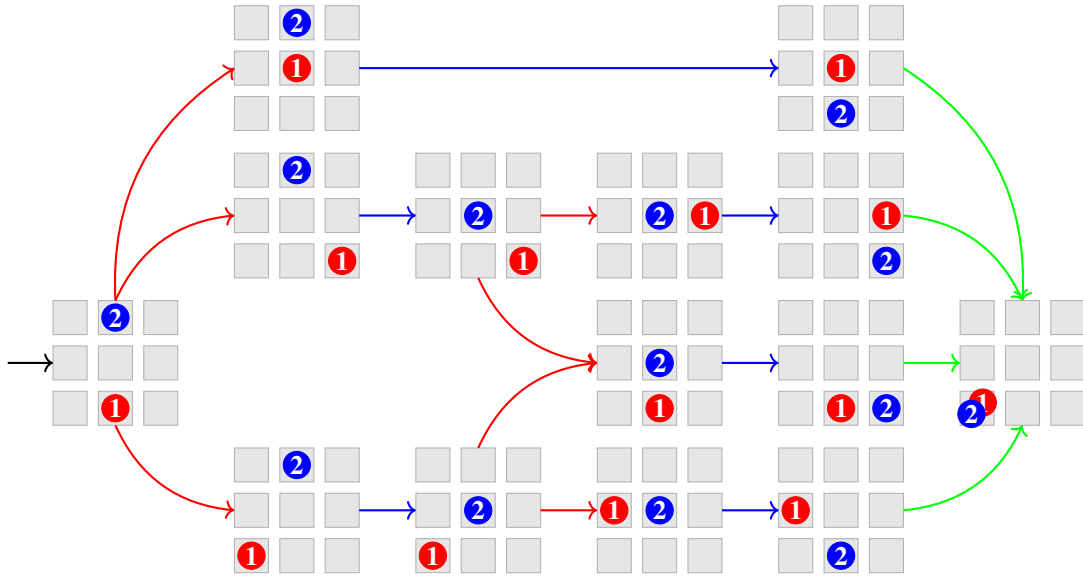


Fig. 16: A graphical representation of the evidence graph of figure 15 using board notation.

6. FUTURE RESEARCH

In this section we discuss some suggestions on future research that can be considered.

6.1 Improving the Efficiency of the Model

According to the rules of the game, a fence may not block the goal line for any of the players. Hence, to ensure that this rule is not violated, we need to check whether the goal lines for both players on the board with the additional fence would still be reachable. In order to check whether the goal line remains accessible, we currently use an algorithm that is based on the depth first search algorithm. In each turn, we have to evaluate for all non-blocked fence positions, whether it would be allowed to play a fence at that location. Hence this method of checking the fence locations is very time consuming. We have already implemented two improvements to increase the efficiency of this procedure, that is, we only do these reachability checks whenever a fence touches two or more fences, or 1 fence and the border of the board. Furthermore, we keep track of blocked fence positions, which prevents even more reachability checks. In future research one could look into improving the efficiency of this procedure, reducing the computational complexity of the model.

6.2 Formulating and Generalizing Winning Strategies

The results published in this paper show that there exist winning strategies for smaller instances of the game. Evidence graphs have been generated that show the moves of the winning player. Using these graphs, a strategy can be formulated as a number of guidelines on how to play the game in an optimal manner, such that a win is guaranteed. It could be possible that the strategies which are obtained by analyzing these evidence graphs can be generalized to strategies that would also guarantee a win for that player in larger instances of the game.

7. REFERENCES

- [1] Jonathan Schaeffer et al. “Checkers Is Solved”. In: *Science* 317.5844 (2007), pp. 1518–1522. DOI: 10.1126/science.1144079.
- [2] Gijs Kant et al. “LTSmin: High-Performance Language-Independent Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 978-3-662-46681-0.
- [3] Gijs Kant and Jaco van de Pol. “Generating and Solving Symbolic Parity Games”. In: *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2014, Grenoble, France, 5th April 2014*. Ed. by Dragan Bosnacki et al. Vol. 159. EPTCS. 2014, pp. 2–14. DOI: 10.4204/EPTCS.159.2.
- [4] Jos W.H.M. Uiterwijk, H. Jaap van den Herik, and L. Victor Allis. “A knowledge-based approach to connect-four: The game is solved!”. In: *Heuristic programming in artificial intelligence*. Ed. by David N.L. Levy and Don F. Beal. Vol. 1. Ellis Horwood series in artificial intelligence. 1989, pp. 113–133. ISBN: 074580778X.
- [5] John W. Romein and Henri E. Bal. “Solving the Game of Awari using Parallel Retrograde Analysis”. English. In: *Computer* 38.10 (2003), pp. 26–33. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1236468.
- [6] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and analysis of communicating systems*. English. MIT Press, 2014. ISBN: 978-0-262-02771-7.
- [7] Roberto Cavada et al. “The nuXMV Symbolic Model Checker”. In: *CAV*. 2014, pp. 334–342.
- [8] Armin Biere and Roderick Bloem, eds. *Computer Aided Verification - 26th International Conference, CAV 2014*,

Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014. ISBN: 978-3-319-08866-2.

- [9] Gerard J. Holzmann. *SPIN Model Checker*. URL: <https://spinroot.com/spin/whatispin.html>.
- [10] UPPAAL. URL: <https://uppaal.org/>.
- [11] Mirko Marchesi. *Quoridor. Gigamic, 1997*.
- [12] Olav Bunte et al. “The mCRL2 Toolset for Analysing Concurrent Systems”. In: *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2019, LNCS vol. 11428*. Cham: Springer International Publishing, 2019, pp. 21–39. ISBN: 978-3-030-17465-1.
- [13] Jan A. Bergstra and Jan W. Klop. “Process algebra for synchronous communication”. In: *Information and Control* 60.1 (1984), pp. 109–137. ISSN: 0019-9958. DOI: [https://doi.org/10.1016/S0019-9958\(84\)80025-X](https://doi.org/10.1016/S0019-9958(84)80025-X).
- [14] L. Victor Allis. “Searching for solutions in games and artificial intelligence”. English. PhD thesis. Maastricht University, Jan. 1994. ISBN: 9090074880. DOI: 10.26481/dis.19940923la.
- [15] John F. Nash. *Some Games and Machines for Playing Them*. Santa Monica, CA: RAND Corporation, 1952.
- [16] Dennis M. Breuker, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. “Solving 8×8 Domineering”. In: *Theoretical Computer Science* 230.1 (2000), pp. 195–206. ISSN: 0304-3975. DOI: [https://doi.org/10.1016/S0304-3975\(99\)00082-1](https://doi.org/10.1016/S0304-3975(99)00082-1).
- [17] Nathan Bullock. “Domineering: Solving Large Combinatorial Search Spaces”. In: *J. Int. Comput. Games Assoc.* 25 (2002), pp. 67–84.
- [18] Jos W. H. M. Uiterwijk. “11 × 11 Domineering Is Solved: The First Player Wins”. In: *Computers and Games*. Ed. by Aske Plaatt, Walter Kisters, and Jaap van den Herik. Cham: Springer International Publishing, 2016, pp. 129–136. ISBN: 978-3-319-50935-8.
- [19] Maarten P.D. Schadd et al. “Best Play in Fanorona Leads to Draw”. In: *New Mathematics and Natural Computation* 04.03 (2008), pp. 369–387. DOI: 10.1142/S1793005708001124.
- [20] Ralph Gasser. “Solving Nine Men’s Morris”. In: *Computational Intelligence* 12.1 (1996), pp. 24–41. DOI: <https://doi.org/10.1111/j.1467-8640.1996.tb00251.x>.
- [21] Oren Patashnik. “Qubic: 4×4×4 Tic-Tac-Toe”. In: *Mathematics Magazine* 53 (1980), pp. 202–216.
- [22] John Wagner and István C. Virág. “Solving Renju”. In: *J. Int. Comput. Games Assoc.* 24 (2001), pp. 30–35.
- [23] John Tromp. *John’s Connect Four Playground*. Dec. 2014. URL: <https://tromp.github.io/c4/c4.html>.
- [24] Geoffrey Irving, Jeroen H.H.L.M. Donkers, and Jos W.H.M. Uiterwijk. “Solving Kalah”. English. In: *ICGA Journal* 23 (Jan. 2000), pp. 139–147. ISSN: 1389-6911. DOI: 10.3233/ICG-2000-23303.
- [25] Jan Friso Groote and Erik P. de Vink. “Problem Solving Using Process Algebra Considered Insightful”. In: *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*. Ed. by Joost-Pieter Katoen, Rom Langerak, and Arend Rensink. Cham: Springer International Publishing, 2017, pp. 48–63. ISBN: 978-3-319-68270-9. DOI: 10.1007/978-3-319-68270-9_3.
- [26] James D. Allen. *The Complete Book of Connect 4: History, Strategy, Puzzles*. Sterling Publishing Company, Incorporated, 2010. ISBN: 9781402756214. URL: <https://books.google.nl/books?id=3LqRQQAACAAJ>.
- [27] L. Victor Allis. “A Knowledge-Based Approach of Connect-Four”. In: *J. Int. Comput. Games Assoc.* 11 (1988).
- [28] John Tromp. Jan. 2017. URL: <https://github.com/tromp/tromp.github.io/blob/master/c4/icga.pdf>.
- [29] Michael Baldamus et al. “Can American Checkers be Solved by Means of Symbolic Model Checking?” In: *Electronic Notes in Theoretical Computer Science* 43 (2001), pp. 15–33. ISSN: 1571-0661. DOI: [https://doi.org/10.1016/S1571-0661\(04\)80892-2](https://doi.org/10.1016/S1571-0661(04)80892-2).
- [30] *Game complexity: Complexities of some well-known games*. Dec. 2022. URL: https://en.wikipedia.org/wiki/Game_complexity#Complexities_of_some_well-known_games.
- [31] *mCRL2 GitHub Repository: Game Examples*. URL: <https://github.com/mCRL2org/mCRL2/tree/master/examples/games>.
- [32] Lisa Glendenning. “Mastering Quoridor”. PhD thesis. 2005. URL: https://www.dropbox.com/s/uhkgkfrfrdaecpv/glendenning_ugrad_thesis.pdf.
- [33] Peter J. C. Mertens. “A Quoridor-playing Agent”. In: 2006.
- [34] Guy Tsur and Yotam Segev. 2012. URL: <https://www.cs.huji.ac.il/w~ai/projects/2012/Quoridor/files/report.pdf>.
- [35] Victor M. Respall, Joseph Brown, and Hamna Aslam. “Monte Carlo Tree Search for Quoridor”. In: Sept. 2018.
- [36] Chris Jose et al. *Quoridor-AI*. Sept. 2022.

APPENDIX A

QUORIDOR MCRL2 MODEL

```
1 sort
2   Position = struct pos(col: Pos, row: Pos);
3   FPosition = struct fpos(col: Pos, row: Pos, dir: Direction);
4   Direction = struct H | V;
5   Turn = struct P1 | P2 | None;
6
7 map
8   % N is the number of squares on 1 row/column of the board, the board is N by N squares
9   N: Pos;
10
11   % F is the number of fences per player. Therefore, in total 2F fences can be used throughout the game
12   F: Nat;
13
14   % Get center column
15   upperMiddle: Pos;
16
17   % Check whether a position is valid
18   validPos: Position -> Bool;
19
20   % Check whether a fence is placed between two coordinates
21   isBlocked: Position # Position # FSet(FPosition) -> Bool;
22
23   % Check whether a move is valid, format: curPos, opponentPos, newPos, Fences -> Bool valid
24   isValidMove: Position # Position # Position # FSet(FPosition) -> Bool;
25
26   % Check whether the placement of a fence is valid
27   isValidPlace: Position # Position # FPosition # FSet(FPosition) -> Bool;
28
29   % Check whether the goal of each player remains reachable
30   isGoalReachable: Turn # List(Position) # FSet(Position) # FSet(FPosition) -> Bool;
31
32   % Check how many fences the requested fence touches.
33   countFenceTouching: FPosition # FSet(FPosition) -> Nat;
34
35   % Add blocked fences
36   addBlocked: FSet(FPosition) # FPosition -> FSet(FPosition);
37
38 var
39   p : Turn;
40   pA, pB, pD : Position;
41   fences : FSet(FPosition);
42   lv : List(Position);
43   lp : FSet(Position);
44   fp : FPosition;
45
46 eqn
47   N = 3;
48   F = 1;
49
50   upperMiddle = Int2Pos(N div 2);
51
52   % A position pA is valid if both its column and row are in range 1..N
53   validPos(pA) = 1 <= col(pA) && 1 <= row(pA) && col(pA) <= N && row(pA) <= N;
54
55   % A move from position pA to position pD is blocked if pD is invalid (not on the board)
56   % or if there is a fence between pA and pD.
57   isBlocked(pA, pD, fences) = !validPos(pD) || if (col(pA) != col(pD),
58     % Adjacent cells in same row, different column
59     (
60       fpos(min(col(pA), col(pD)), row(pA), V) in fences
61       || if (row(pA) == 1, false, fpos(min(col(pA), col(pD)), Int2Pos(row(pA) - 1), V) in fences)
62     ),
63     % Adjacent cells in same column, different row
64     (
65       fpos(col(pA), min(row(pA), row(pD)), H) in fences
66       || if (col(pA) == 1, false, fpos(Int2Pos(col(pA) - 1), min(row(pA), row(pD)), H) in fences)
67     )
68   );
69
70   % Calculate whether the move from pA to pD is valid, pB is the location of the opponent.
71   % fences is the list of fences which are placed on the board.
72   isValidMove(pA, pB, pD, fences) = if(
73     % A valid move either consist of 1 or 2 steps, the destination may not be occupied and should be a
74     % valid position. If any of these conditions are violated, we can directly return false.
75     (
76       (abs(col(pA) - col(pD)) + abs(row(pA) - row(pD)) > 2)
77       || (abs(col(pA) - col(pD)) + abs(row(pA) - row(pD)) == 0)
78       || pB == pD
79       || !validPos(pD)
80     ),
```

```

81 % Not valid according to above mentioned conditions
82 false ,
83
84 % So far, it is valid , we check if it is a 1 step move
85 if (
86     % 1 step move
87     (abs(col(pA) - col(pD)) + abs(row(pA) - row(pD)) == 1),
88
89     % We only need to check whether there is no fence blocking the move
90     !isBlocked(pA, pD, fences),
91
92     % 2 step move
93     if (
94         % Determine whether its a straight or diagonal jump
95         (col(pA) == col(pD) || row(pA) == row(pD)),
96
97         % Jump over opponent in line (not diagonal)
98         (
99             (
100                 (pB == pos(min(col(pA), col(pD)) + 1, row(pA)) && row(pA) == row(pD))
101                 || (pB == pos(col(pA), min(row(pA), row(pD)) + 1) && col(pA) == col(pD))
102             )
103             && !isBlocked(pA, pB, fences)
104             && !isBlocked(pB, pD, fences)
105         ),
106
107         % Diagonal Jump over opponent
108         (
109             (pB == pos(col(pA), row(pD)) || pB == pos(col(pD), row(pA)))
110             && !isBlocked(pA, pB, fences)
111             && !isBlocked(pB, pD, fences)
112             && if (
113                 % Check whether the opponent is in the same column
114                 pB == pos(col(pA), row(pD)),
115
116                 % Opponent is in the same column
117                 if (row(pB) - (row(pA)-row(pB)) < 1 || row(pB) - (row(pA)-row(pB)) > N, true ,
118                     isBlocked(pB, pos(col(pA), Int2Pos(row(pB) - (row(pA)-row(pB)))), fences)
119                 ),
120
121                 % Opponent is in the same row
122                 if (col(pB) - (col(pA)-col(pB)) < 1 || col(pB) - (col(pA)-col(pB)) > N, true ,
123                     isBlocked(pB, pos(Int2Pos(col(pB) - (col(pA)-col(pB))), row(pA)), fences)
124                 )
125             )
126         )
127     )
128 )
129 );
130
131 % A place is valid if it is not overlapping another fence and the goals remain reachable
132 isValidPlace(pA, pB, fp, fences) = if (
133     % We check if it is touching 2 fences or 1 fence and the border of the board
134     % If so, we need to check if it blocks the goal line for both players
135     (
136         X == 1 && ((dir(fp) == H && (col(fp) == 1 || col(fp) == N - 1))
137         || (dir(fp) == V && (row(fp) == 1 || row(fp) == N - 1)))
138         || X >= 2,
139     isGoalReachable(P1, [pA], {}, {fp} + fences) && isGoalReachable(P2, [pB], {}, {fp} + fences), true)
140     while X = countFenceTouching(fp, fences) end;
141
142 % addBlocked adds all fences to the fset that would overlap with the placed fence
143 addBlocked(fences, fp) =
144     {fp}
145     + {fpos(col(fp), row(fp), if(dir(fp) == V, H, V))}
146     + if((col(fp) == 1 && dir(fp) == H) || (row(fp) == 1 && dir(fp) == V), {},
147         {fpos(Int2Pos(col(fp) - if(dir(fp) == H, 1, 0)), Int2Pos(row(fp) - if(dir(fp) == V, 1, 0)), dir(fp))})
148     + {fpos(col(fp) + if(dir(fp) == H, 1, 0), row(fp) + if(dir(fp) == V, 1, 0), dir(fp))}
149     + fences;
150
151 % isGoalReachable checks if the goal line for a player is still reachable.
152 % The arguments are: the player for which we check (Turn), a list of visited but not processed positions
153 % (initially the position of the player), the list of processed positions (initially empty) and
154 % the set of fences. This function follows an BFS approach.
155 isGoalReachable(p, [], lp, fences) = (exists g : Position . g in lp && row(g) == if(p == P1, N, 1));
156 isGoalReachable(None, lv, lp, fences) = true;
157 (p == P1 || p == P2) -> isGoalReachable(p, pA |> lv, lp, fences) =
158     if (
159         % Always allowed if its the first fence placed or if player has reached goal
160         # fences == 1 || row(pA) == if(p == P1, N, 1),
161
162         % Condition holds
163

```

```

164         true ,
165
166         % Add all 4 adjacent squares to list if that move is not blocked
167         isGoalReachable(p,
168             if(
169                 % Condition
170                 !(pos(col(pA), row(pA)+1) in lv || pos(col(pA), row(pA)+1) in lp)
171                 && validPos(pos(col(pA), row(pA)+1))
172                 && !isBlocked(pA, pos(col(pA), row(pA)+1), fences),
173
174                 % Condition holds
175                 [pos(col(pA), row(pA)+1)],
176
177                 % Condition does not hold
178                 []
179             ) ++
180             if (row(pA) <= 1, [],
181                 if(
182                     % Condition
183                     !(pos(col(pA), Int2Pos(row(pA)-1)) in lv || pos(col(pA), Int2Pos(row(pA)-1)) in lp)
184                     && validPos(pos(col(pA), Int2Pos(row(pA)-1)))
185                     && !isBlocked(pA, pos(col(pA), Int2Pos(row(pA)-1)), fences),
186
187                     % Condition holds
188                     [pos(col(pA), Int2Pos(row(pA)-1))],
189
190                     % Condition does not hold
191                     []
192                 )
193             ) ++
194             if (col(pA) <= 1, [],
195                 if(
196                     % Condition
197                     !(pos(Int2Pos(col(pA)-1), row(pA)) in lv || pos(Int2Pos(col(pA)-1), row(pA)) in lp)
198                     && validPos(pos(Int2Pos(col(pA)-1), row(pA)))
199                     && !isBlocked(pA, pos(Int2Pos(col(pA)-1), row(pA)), fences),
200
201                     % Condition holds
202                     [pos(Int2Pos(col(pA)-1), row(pA))],
203
204                     % Condition does not hold
205                     []
206                 )
207             ) ++
208                 if(
209                     % Condition
210                     !(pos(col(pA)+1, row(pA)) in lv || pos(col(pA)+1, row(pA)) in lp)
211                     && validPos(pos(col(pA)+1, row(pA)))
212                     && !isBlocked(pA, pos(col(pA)+1, row(pA)), fences),
213
214                     % Condition holds
215                     [pos(col(pA)+1, row(pA))],
216
217                     % Condition does not hold
218                     []
219                 ) ++
220                 lv, {pA} + lp, fences)
221         );
222
223 % Check if the fence placed is touching another fence on the board.
224 % Used to reduce the number of isGoalReachable executions
225 countFenceTouching(fp, fences) =
226     if(dir(fp) == H,
227         % Horizontal placed fence
228         if(col(fp) <= 2, 0, if(fpos(Int2Pos(col(fp) - 2), row(fp), dir(fp)) in fences, 1, 0))
229         + if(fpos(col(fp) + 2, row(fp), dir(fp)) in fences, 1, 0),
230
231         % Vertical placed fence
232         if(row(fp) <= 2, 0, if(fpos(col(fp), Int2Pos(row(fp) - 2), dir(fp)) in fences, 1, 0))
233         + if(fpos(col(fp), row(fp) + 2, dir(fp)) in fences, 1, 0)
234     )
235     + if((fpos(col(fp) + 1, row(fp) + 1, if(dir(fp) == V, H, V)) in fences), 1, 0)
236     + if((fpos(col(fp), row(fp) + 1, if(dir(fp) == V, H, V)) in fences), 1, 0)
237     + if(col(fp) <= 1, 0, if(fpos(Int2Pos(col(fp) - 1), row(fp) + 1, if(dir(fp) == V, H, V)) in fences, 1, 0))
238     + if((fpos(col(fp) + 1, row(fp), if(dir(fp) == V, H, V)) in fences), 1, 0)
239     + if(col(fp) <= 1, 0, if(fpos(Int2Pos(col(fp) - 1), row(fp), if(dir(fp) == V, H, V)) in fences, 1, 0))
240     + if(row(fp) <= 1, 0, if(fpos(col(fp) + 1, Int2Pos(row(fp) - 1), if(dir(fp) == V, H, V)) in fences, 1, 0))
241     + if(row(fp) <= 1, 0, if(fpos(col(fp), Int2Pos(row(fp) - 1), if(dir(fp) == V, H, V)) in fences, 1, 0))
242     + if(col(fp) <= 1 || row(fp) <= 1, 0,
243         if(fpos(Int2Pos(col(fp) - 1), Int2Pos(row(fp) - 1), if(dir(fp) == V, H, V)) in fences, 1, 0));
244
245 act
246 movePawn: Turn # Pos # Pos;

```

```

247 addFence: Turn # Pos # Pos # Direction;
248 win: Turn;
249
250 proc Game(turn:Turn, p1:Position, p2:Position, rf1:Nat, fences:FSet(FPosition), blocked:FSet(FPosition)) =
251   (turn != None) => (
252     (row(p1) == N || row(p2) == 1)
253     % One of the players won
254     => win(if(row(p1) == N, P1, P2)).Game(None, pos(1,1), pos(1,1), 0, {}, {})
255     % No winning player yet
256     <> (
257       (turn == P1)
258       => (
259         % Player 1 moves his pawn
260         (sum c,r:Pos.
261           (validPos(pos(c, r)) && isValidMove(p1, p2, pos(c, r), fences))
262           => movePawn(turn, c, r).Game(P2, pos(c, r), p2, rf1, fences, blocked)
263         )
264         % Player 1 places a fence on the board
265         + (sum c,r:Pos.sum d:Direction.(
266           validPos(pos(c, r)) && validPos(pos(c + 1, r + 1))
267           && !(fpos(c, r, d) in blocked)
268           && rf1 > 0 && isValidPlace(p1, p2, fpos(c, r, d), fences)
269         ) => addFence(turn, c, r, d)
270           .Game(P2, p1, p2, Int2Nat(rf1 - 1), {fpos(c, r, d)} + fences, addBlocked(blocked, fpos(c,r,d))))
271       )
272     + (turn == P2)
273     => (
274       % Player 2 moves his pawn
275       (sum c,r:Pos.
276         (validPos(pos(c, r)) && isValidMove(p2, p1, pos(c, r), fences))
277         => movePawn(turn, c, r).Game(P1, p1, pos(c, r), rf1, fences, blocked)
278       )
279       % Player 2 places a fence on the board
280       + (sum c,r:Pos.sum d:Direction.(
281         validPos(pos(c, r)) && validPos(pos(c + 1, r + 1))
282         && !(fpos(c, r, d) in blocked)
283         && ((2 * F) - (# fences) - rf1) > 0 && isValidPlace(p1, p2, fpos(c, r, d), fences)
284       ) => addFence(turn, c, r, d)
285         .Game(P1, p1, p2, rf1, {fpos(c, r, d)} + fences, addBlocked(blocked, fpos(c,r,d))))
286     )
287   )
288 );
289
290 init
291   allow ({
292     movePawn, addFence, win
293   },
294   Game(P1, pos(upperMiddle + 1,1), pos(upperMiddle + 1,N), F, {}, {}))
295 );

```


APPENDIX B

MODAL μ -CALCULUS FORMULAE

B.1 Rule 1

```

1 nu X(rfP1:Nat=F, rfP2:Nat=F). (
2   (forall r,c:Pos. forall d:Direction. [addFence(P1, c, r, d)] (val(rfP1 > 0) && X(Int2Nat(rfP1 - 1), rfP2))) &&
3   (forall r,c:Pos. forall d:Direction. [addFence(P2, c, r, d)] (val(rfP2 > 0) && X(rfP1, Int2Nat(rfP2 - 1)))) &&
4   [!(exists r,c:Pos. exists d:Direction. exists t:Turn. addFence(t, c, r, d))] X(rfP1, rfP2)
5 )

```

B.2 Rule 3

```

1 [!(exists c1,r1:Pos. exists d1:Direction. movePawn(P1, c1, r1) || addFence(P1, c1, r1, d1))*.
2   forall c2,r2:Pos. (val(c2 <= N) && val(r2 <= N)) => forall d2:Direction.
3   (movePawn(P2, c2, r2) || addFence(P2, c2, r2, d2))
4 ] false

```

B.3 Rule 4

```

1 nu X(turn:Turn=P1). (
2   forall r,c:Pos. (val(c <= N) && val(r <= N)) =>
3   forall t:Turn. [movePawn(t, c, r)] (val(t == turn) && X(if(t == P1, P2, P1))) &&
4   forall r,c:Pos. (val(c <= N) && val(r <= N)) =>
5   forall t:Turn. forall d:Direction. [addFence(t, c, r, d)] (val(t == turn) && X(if(t == P1, P2, P1))) &&
6   [!(exists r,c:Pos. (val(c <= N) && val(r <= N)) =>
7     exists t:Turn. exists d:Direction. (movePawn(t,c,r) || addFence(t, c, r, d)))] X(turn)
8 )

```

B.4 Rule 5

```

1 nu X(turn:Turn=P1, rfP1:Nat=F, rfP2:Nat=F, fences:FSet(FPosition)={}.pA:Position=pos(upperMiddle + 1, 1).pB:Position=pos(upperMiddle + 1, N)). (
2   [!(exists r,c:Pos. exists d:Direction. exists t:Turn. movePawn(t, c, r) || addFence(t, c, r, d))] X(turn, rfP1, rfP2, fences, pA, pB)) &&
3   (val(turn == P1) => (exists c,r:Pos. exists d:Direction.
4     <movePawn(P1, c, r)>X(if(r==N, None, P2), rfP1, rfP2, fences, pos(c,r), pB) &&
5     (exists c3,r3:Pos. exists d3:Direction. (val(rfP1 > 0) && val(isValidPlace(pA, pB, fpos(c3, r3, d3), fences))) =>
6       <addFence(P1, c3, r3, d3)>X(P2, Int2Nat(rfP1 - 1), rfP2, {fpos(c3, r3, d3)} + fences, pA, pB))
7   )) &&
8   (val(turn == P2) => (exists c,r:Pos. exists d:Direction.
9     <movePawn(P2, c, r)>X(if(r==1, None, P1), rfP1, rfP2, fences, pA, pos(c,r)) &&
10    (exists c3,r3:Pos. exists d3:Direction. (val(rfP2 > 0) && val(isValidPlace(pA, pB, fpos(c3, r3, d3), fences))) =>
11      <addFence(P2, c3, r3, d3)>X(P1, rfP1, Int2Nat(rfP2 - 1), {fpos(c3, r3, d3)} + fences, pA, pB))
12   ))
13 )
14 )
15 )

```

B.5 Rule 6a

```

1 nu X(fences:FSet(FPosition)={}.pA:Position=pos(upperMiddle + 1, 1).pB:Position=pos(upperMiddle + 1, N)). (
2   (forall c,r:Pos. (val(c <= N) && val(r <= N)) &&
3     (
4       (val(abs(col(pA)-c)+abs(row(pA)-r) == 1) && val(!isBlocked(pA, pos(c,r), fences)))
5       || (
6         val(abs(col(pA)-c)+abs(row(pA)-r) == 2)
7         && val(col(pA) != c)
8         && val(row(pA) != r)
9         && val(!isBlocked(pA, pB, fences))
10        && val(!isBlocked(pB, pos(c,r), fences))
11        && (
12          val(if(pB == pos(col(pA), r),
13            if (row(pB) - (row(pA)-row(pB)) < 1 || row(pB) - (row(pA)-row(pB)) > N, true,
14              isBlocked(pB, pos(col(pA), Int2Pos(row(pB) - (row(pA)-row(pB)))), fences)
15          ),
16          if (col(pB) - (col(pA)-col(pB)) < 1 || col(pB) - (col(pA)-col(pB)) > N, true,
17            isBlocked(pB, pos(Int2Pos(col(pB) - (col(pA)-col(pB))), row(pA)), fences)
18          )
19        )
20      )
21    )
22    || (
23      val(abs(col(pA)-c)+abs(row(pA)-r) == 2)
24      && (val(col(pA) == c) || val(row(pA) != r))
25      && val(!isBlocked(pA, pB, fences))
26      && val(!isBlocked(pB, pos(c,r), fences))
27    )
28  ) =>
29  (
30    [movePawn(P1, c, r)] (val(!isBlocked(pA, pos(c,r), fences)) && X(fences, pos(c, r), pB)) ||
31    [movePawn(P2, c, r)] (val(!isBlocked(pB, pos(c,r), fences)) && X(fences, pA, pos(c, r)))
32  )
33 ) &&
34 (val(# fences < 2 * F) => (forall c,r:Pos. (val(c <= N) && val(r <= N)) => forall d:Direction.
35   [addFence(P1, c,r,d) || addFence(P2, c,r,d)] X({fpos(c,r,d)} + fences, pA, pB))) &&
36 [!(exists c,r:Pos. exists d:Direction. val(c <= N) && val(r <= N) && (
37   movePawn(P1, c, r) || movePawn(P2, c, r) || addFence(P1,c,r,d) || addFence(P2,c,r,d)
38 ))] X(fences, pA, pB)
39 )

```

B.6 Rule 6b

```
1 nu X(pA:Position=pos(upperMiddle + 1, 1), pB:Position=pos(upperMiddle + 1, N)).(  
2   forall c,r:Pos.[movePawn(P1, c, r)] (val(col(pB) != c || row(pB) != r) && X(pos(c,r), pB)) &&  
3   forall c,r:Pos.[movePawn(P2, c, r)] (val(col(pA) != c || row(pA) != r) && X(pA, pos(c,r))) &&  
4   [!(exists c,r:Pos.(movePawn(P1, c, r) || movePawn(P2, c, r)))] X(pA, pB)  
5 )
```

B.7 Rule 7, 8 & 9

```
1 nu X(pA:Position=pos(upperMiddle + 1, 1), pB:Position=pos(upperMiddle + 1, N), fences:FSet(FPosition)={}).(  
2   forall c,r:Pos.(val(c <= N) && val(r <= N)) => [movePawn(P1, c, r)] (  
3     (  
4       % Move to adjacent place  
5       (val(c == col(pA) + 1) && val(r == row(pA)))  
6       || (val(c == col(pA) - 1) && val(r == row(pA)))  
7       || (val(c == col(pA)) && val(r == row(pA) + 1))  
8       || (val(c == col(pA)) && val(r == Int2Pos(row(pA) - 1)))  
9       % Jump in straight line  
10      || (val(c == col(pA) + 2) && val(r == row(pA)) && val(col(pA) + 1 == col(pB)) && val(row(pA) == row(pB)))  
11      || (val(c == Int2Pos(col(pA) - 2)) && val(r == row(pA)) && val(Int2Pos(col(pA) - 1) == col(pB)) && val(row(pA) == row(pB)))  
12      || (val(c == col(pA)) && val(r == row(pA) + 2) && val(col(pA) == col(pB)) && val(row(pA) + 1 == row(pB)))  
13      || (val(c == col(pA)) && val(r == Int2Pos(row(pA) - 2)) && val(col(pA) == col(pB)) && val(Int2Pos(row(pA) - 1) == row(pB)))  
14      % Diagonal Jump  
15      || (val(c == col(pA) + 1) && val(r == row(pA) + 1) &&  
16        (  
17          val(isBlocked(pB, pos(col(pA) + 2, row(pA)), fences) && col(pA) + 1 == col(pB) && row(pA) == row(pB))  
18          || val(isBlocked(pB, pos(col(pA), row(pA) + 2), fences) && col(pA) == col(pB) && row(pA) + 1 == row(pB))  
19        )  
20      )  
21      || val(if(row(pA) == 1, false, ((c == col(pA) + 1) && (r == Int2Pos(row(pA) - 1)) &&  
22        (  
23          (isBlocked(pB, pos(col(pA) + 2, row(pA)), fences) && col(pA) + 1 == col(pB) && row(pA) == row(pB))  
24          || (if((row(pA) - 2 < 1), true, isBlocked(pB, pos(col(pA), Int2Pos(row(pA) - 2)), fences)) && col(pA) == col(pB) && Int2Pos(row(pA) - 1) == row(pB))  
25        )  
26      )))  
27      || val(if((row(pA) == 1) || (col(pA) == 1), false, ((c == Int2Pos(col(pA) - 1)) && (r == Int2Pos(row(pA) - 1)) &&  
28        (  
29          (if((col(pA) - 2 < 1), true, isBlocked(pB, pos(Int2Pos(col(pA) - 2), row(pA)), fences)) && Int2Pos(col(pA) - 1) == col(pB) && row(pA) == row(pB))  
30          || (if((row(pA) - 2 < 1), true, isBlocked(pB, pos(col(pA), Int2Pos(row(pA) - 2)), fences)) && col(pA) == col(pB) && Int2Pos(row(pA) - 1) == row(pB))  
31        )  
32      )))  
33      || val(if((col(pA) == 1), false, ((c == Int2Pos(col(pA) - 1)) && (r == row(pA) + 1) &&  
34        (  
35          (if((col(pA) - 2 < 1), true, isBlocked(pB, pos(Int2Pos(col(pA) - 2), row(pA)), fences)) && Int2Pos(col(pA) - 1) == col(pB) && row(pA) == row(pB))  
36          || (isBlocked(pB, pos(col(pA), row(pA) + 2), fences) && col(pA) == col(pB) && row(pA) + 1 == row(pB))  
37        )  
38      )))  
39    ) && X(pos(c,r), pB, fences)) &&  
40    forall c,r:Pos.(val(c <= N) && val(r <= N)) => [movePawn(P2, c, r)] (  
41      (  
42        % Move to adjacent place  
43        (val(c == col(pB) + 1) && val(r == row(pB)))  
44        || (val(c == Int2Pos(col(pB) - 1)) && val(r == row(pB)))  
45        || (val(c == col(pB)) && val(r == row(pB) + 1))  
46        || (val(c == col(pB)) && val(r == Int2Pos(row(pB) - 1)))  
47        % Jump in straight line  
48        || (val(c == col(pB) + 2) && val(r == row(pB)) && val(col(pB) + 1 == col(pA)) && val(row(pB) == row(pA)))  
49        || (val(c == Int2Pos(col(pB) - 2)) && val(r == row(pB)) && val(Int2Pos(col(pB) - 1) == col(pA)) && val(row(pB) == row(pA)))  
50        || (val(c == col(pB)) && val(r == row(pB) + 2) && val(col(pB) == col(pA)) && val(row(pB) + 1 == row(pA)))  
51        || (val(c == col(pB)) && val(r == Int2Pos(row(pB) - 2)) && val(col(pB) == col(pA)) && val(Int2Pos(row(pB) - 1) == row(pA)))  
52        % Diagonal Jump  
53        || (val(c == col(pB) + 1) && val(r == row(pB) + 1) &&  
54          (  
55            val(isBlocked(pA, pos(col(pB) + 2, row(pB)), fences) && col(pB) + 1 == col(pA) && row(pB) == row(pA))  
56            || val(isBlocked(pA, pos(col(pB), row(pB) + 2), fences) && col(pB) == col(pA) && row(pB) + 1 == row(pA))  
57          )  
58        )  
59        || val(if(row(pB) == 1, false, ((c == col(pB) + 1) && (r == Int2Pos(row(pB) - 1)) &&  
60          (  
61            (isBlocked(pA, pos(col(pB) + 2, row(pB)), fences) && col(pB) + 1 == col(pA) && row(pB) == row(pA))  
62            || (if((row(pB) - 2 < 1), true, isBlocked(pA, pos(col(pB), Int2Pos(row(pB) - 2)), fences)) && col(pB) == col(pA) && Int2Pos(row(pB) - 1) == row(pA))  
63          )  
64        )))  
65        || val(if((row(pB) == 1) || (col(pB) == 1), false, ((c == Int2Pos(col(pB) - 1)) && (r == Int2Pos(row(pB) - 1)) &&  
66          (  
67            (if((col(pB) - 2 < 1), true, isBlocked(pA, pos(Int2Pos(col(pB) - 2), row(pB)), fences)) && Int2Pos(col(pB) - 1) == col(pA) && row(pB) == row(pA))  
68            || (if((row(pB) - 2 < 1), true, isBlocked(pA, pos(col(pB), Int2Pos(row(pB) - 2)), fences)) && col(pB) == col(pA) && Int2Pos(row(pB) - 1) == row(pA))  
69          )  
70        )))  
71        || val(if((col(pB) == 1), false, ((c == Int2Pos(col(pB) - 1)) && (r == row(pB) + 1) &&  
72          (  
73            (if((col(pB) - 2 < 1), true, isBlocked(pA, pos(Int2Pos(col(pB) - 2), row(pB)), fences)) && Int2Pos(col(pB) - 1) == col(pA) && row(pB) == row(pA))  
74            || (isBlocked(pA, pos(col(pB), row(pB) + 2), fences) && col(pB) == col(pA) && row(pB) + 1 == row(pA))  
75          )  
76        )))  
77      ) && X(pA, pos(c,r), fences)) &&  
78      forall t:Turn. forall c,r:Pos. forall d:Direction.[addFence(t, c, r, d)] X(pA, pB, {fpos(c, r, d)} + fences) &&  
79      [!(exists t:Turn. exists c,r:Pos. exists d:Direction.  
80        (val(c <= N) && val(r <= N))  
81        && (  
82          movePawn(P1, c, r) ||  
83          movePawn(P2, c, r) ||  
84          addFence(t, c, r, d)  
85        )  
86      )] X(pA, pB, fences)  
87    )  
88  )
```

B.8 Rule 11

```
1 nu X(If:List(FPosition)=[]).(
2   forall c,r:Pos.forall d:Direction. forall t:Turn.[addFence(t, c, r, d)]
3   (
4     !val(fpos(c, r, V) in If)
5     && !val(fpos(c, r, H) in If)
6     && val(if(((c == 1) && (d == H)) || ((r == 1) && (d == V))), true, !(fpos(Int2Pos(c - if((d==H), 1, 0)), Int2Pos(r - if((d==V), 1, 0)), d) in If))
7     && !val(fpos(c + if((d==H), 1, 0), r + if((d==V), 1, 0), d) in If)
8   )
9   && X(If ++ [fpos(c, r, d)])
10 )
11 && [!(exists c,r:Pos.exists d:Direction.exists t:Turn.addFence(t,c,r,d))] X(If)
12 )
```

B.9 Rule 12

```
1 nu X(pA:Position=pos(upperMiddle + 1, 1),pB:Position=pos(upperMiddle + 1, N), fences:FSet(FPosition)={}).(
2   (forall t:Turn.forall c,r:Pos.[movePawn(t, c, r)] X(if(t==P1, pos(c,r), pA), if(t==P2, pos(c,r), pB), fences)) &&
3   (forall t:Turn.forall c,r:Pos.forall d:Direction.
4     [addFence(t, c, r, d)] (
5       val(isGoalReachable(P1, [pA], {}, {fpos(c,r,d)} + fences))
6       && val(isGoalReachable(P2, [pB], {}, {fpos(c,r,d)} + fences))
7       && X(pA, pB, {fpos(c,r,d)} + fences)
8     )
9   ) &&
10   [!(exists c,r:Pos.exists t:Turn.exists d:Direction.movePawn(t, c, r) || addFence(t, c, r, d))] X(pA, pB, fences)
11 )
```

B.10 Rule 13

```
1 forall t:Turn.forall c:Pos.(val(c <= N)) => [true*.movePawn(t, c, if(t == P1, N, 1))<win(t)>true
```

B.11 Rule 14

```
1 [!(win(P1) || win(P2))*<true>true
```