Bark: An Attributed Graph Grammar Chess Engine

Davide Marincione

June 4, 2024

1 Introduction

Chess is a fairly complex game, with a relatively wide set of rules and a small but non-trivial branching factor (unlike Go, which has very simple rules but huge branching factor). This makes it a good candidate for projects such as Bark. Unlike traditional chess engines, Bark is not programmed in any language: it is instead a set of rules that can be applied to a graph, which is then used to play chess. This makes it a very complex and unwieldy project, as it requires a lot of work to define the rules and the graph structure. In this report, we will describe the rules and the graph structure used in Bark, how their mechanisms differ from a usual chess engine, and we will discuss the challenges and limitations of the project.

A small disclaimer Since this report is dealing with attributed graph grammars, it is full of words like "node" and "edge"; to avoid repetition, we will use the formatting **node** and ***edge*** to decrease the verbosity.

2 Representing the board

Chess cannot be played without a chessboard, therefore the first step in creating Bark was to define a graph structure that could represent the board.

How it is done Fortuitously, a chessboard is composed of 64 squares and, because of that, it can be conveniently be represented as a set of 64-bit integers, where each bit gives the value of a square with respect to specific information of the board. Because of this, a chessboard is usually represented with twelve 64-bit integers, one for each piece type and color. Not only is this representation very compact, but it also allows for very fast operations, as bitwise "magic bitboard" operations are at the core of every modern chess engine.

Bark's nodes Bark, however, does not use this representation. Instead, the board is composed of 64 **square** and 46 **slide_group** nodes. Each **square** is connected to its neighbors via a ***adj*** and to its **slide_group**s via a ***part_of***. Neighbors are defined only to be those **square**s with a common edge, and **slide_group**s are used to represent the groups of squares reachable in a single move by sliding pieces (bishop, rook or queen). Because of this, **slide_group** is actually an abstract class from which **bishop_like_group** and **rook_like_group** inherit. For ease of view and without real impact to the system: **square** is actually an abstract class which is inherited by **white_square** and **black_square**, **bishop_like_group** too is abstract, inherited by **left_group** and **right_group**, and **rook_like_group** is inherited by **ver_group** and **hor_group**. Finally, each **square** has a `file` and `rank` attribute, which are used to represent the position of the square on the board.

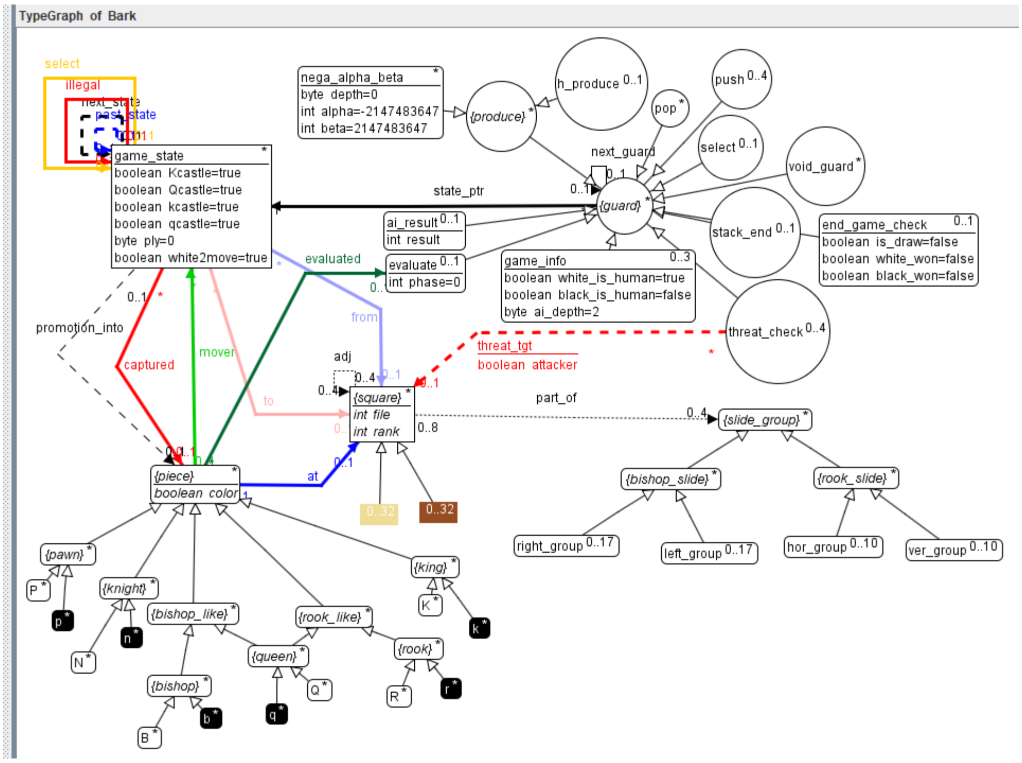

Figure 1: The type graph of the system.

Construction by rules We could have set up the board by hand (as we actually do to position the pieces and starting the game), but that would have been quite tedious because of the multiple edges needed to make it complete. Instead, we use a set of rules to produce it, which are as follows:

1. `produce_first_square` creates a first square of the board (the white top left square, `file=0` and `rank=7`), if it has not been created yet. This is then used as a seed for the rest of the construction.

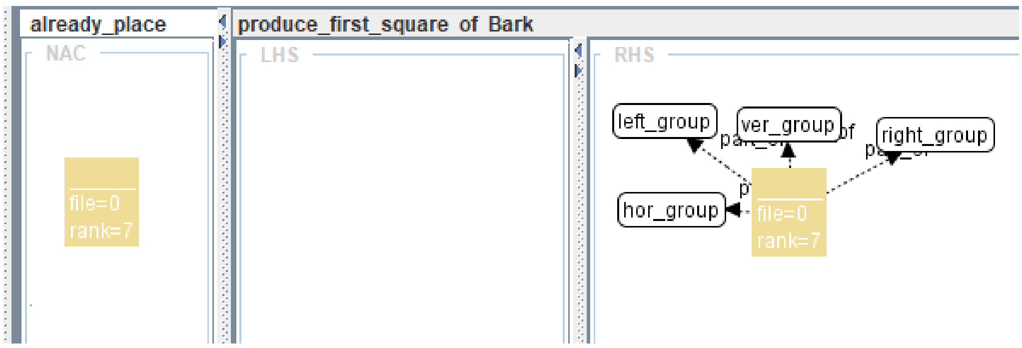

Figure 2: `produce_first_square`

2. `produce_top_rank_b/w` respectively create black and white squares of the top rank, if the they have not been created yet.

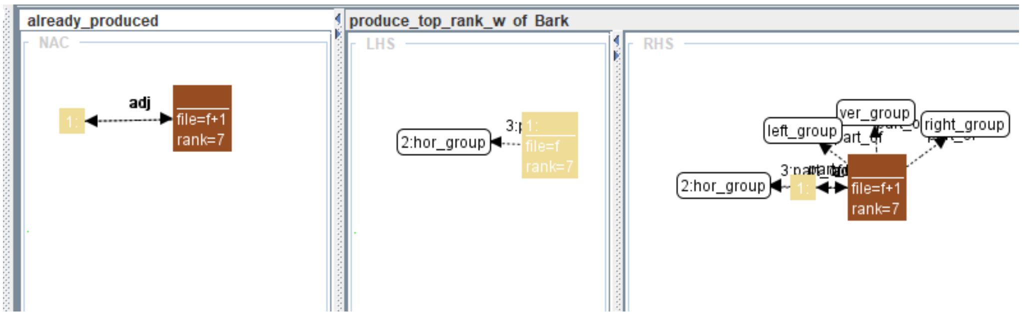

Figure 3: `produce_top_rank`

3. `produce_first_file_b/w` respectively create the black and white squares of the first file, if they have not been created yet.

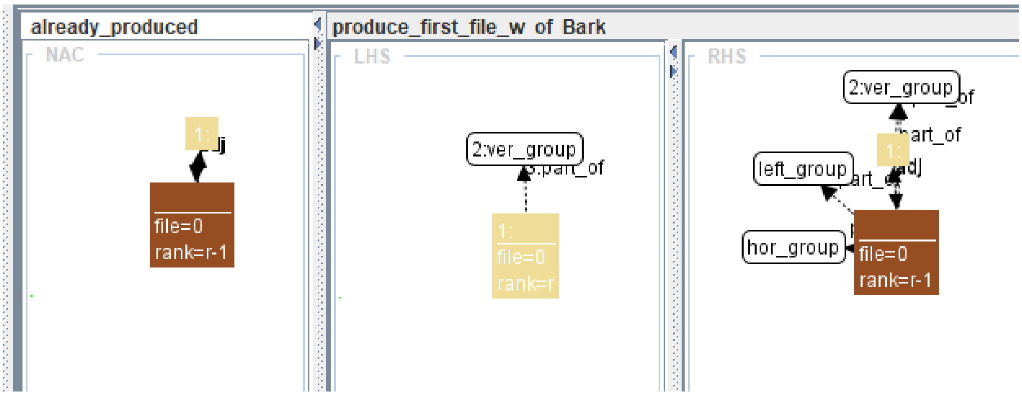

Figure 4: `produce_first_file`

These first rules create the top rank and the left file of the board. The rest of the board is then created by applying:

4. `produce_square_b/w` creates a black or white square, if it has not been created yet.

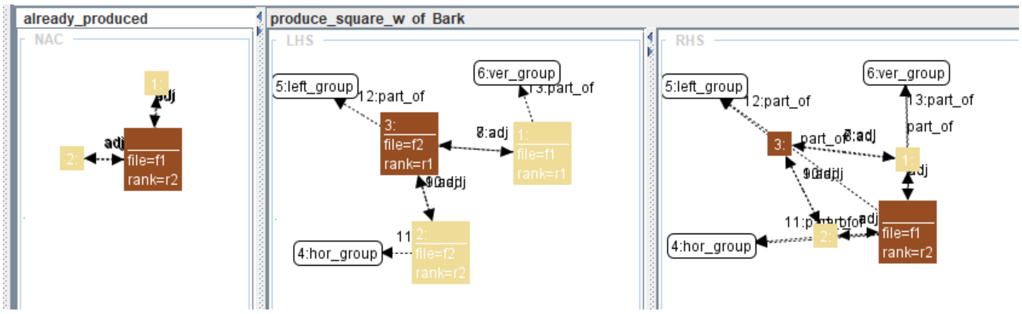

Figure 5: `produce_square`

5. `add_right_group` adds a **right_group** to squares on the last file, if it has not been added yet.

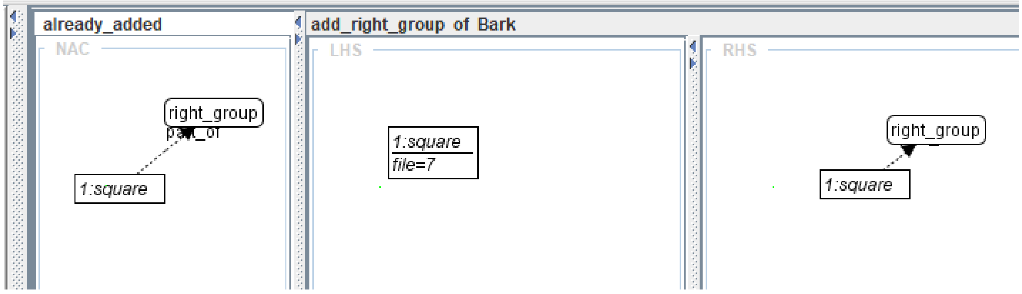

Figure 6: `add_right_group`

6. `produce_right_group` connects top-right to bottom-left the squares to the same group, if it has not been done yet.

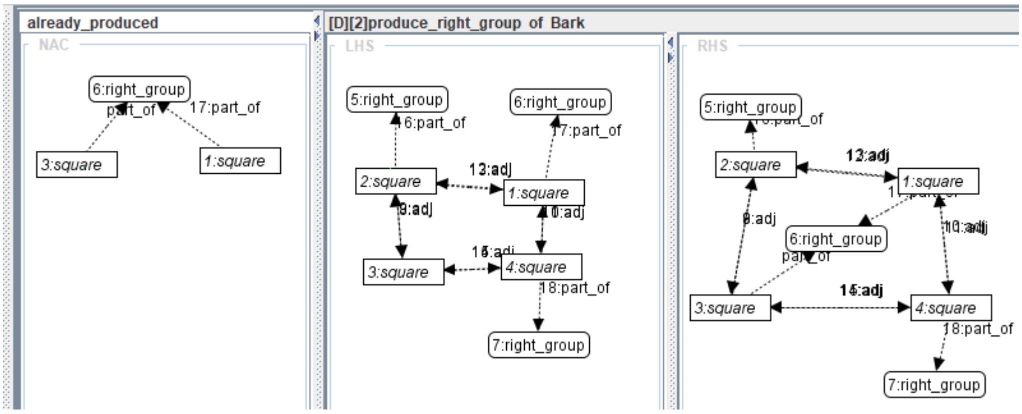

Figure 7: `produce_right_group`

These rules are then applied in a loop until the board is complete. Of course, this is just a *topological* description of the board, and, indeed, applying these rules doesn't result in a good-looking graph.

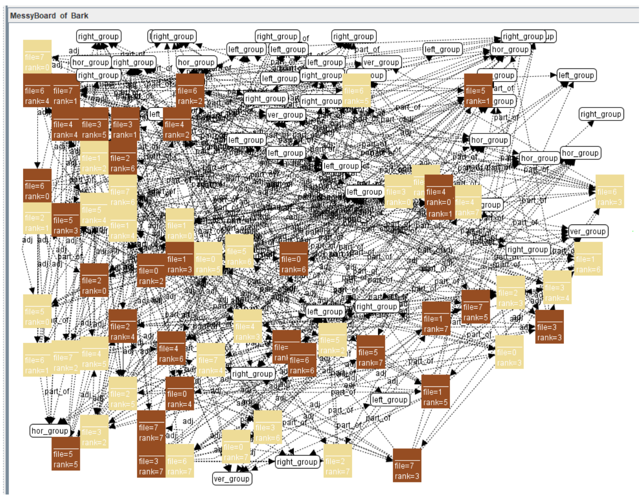

Figure 8: The board after applying the rules.

To make it comprehensible, we order it by hand.

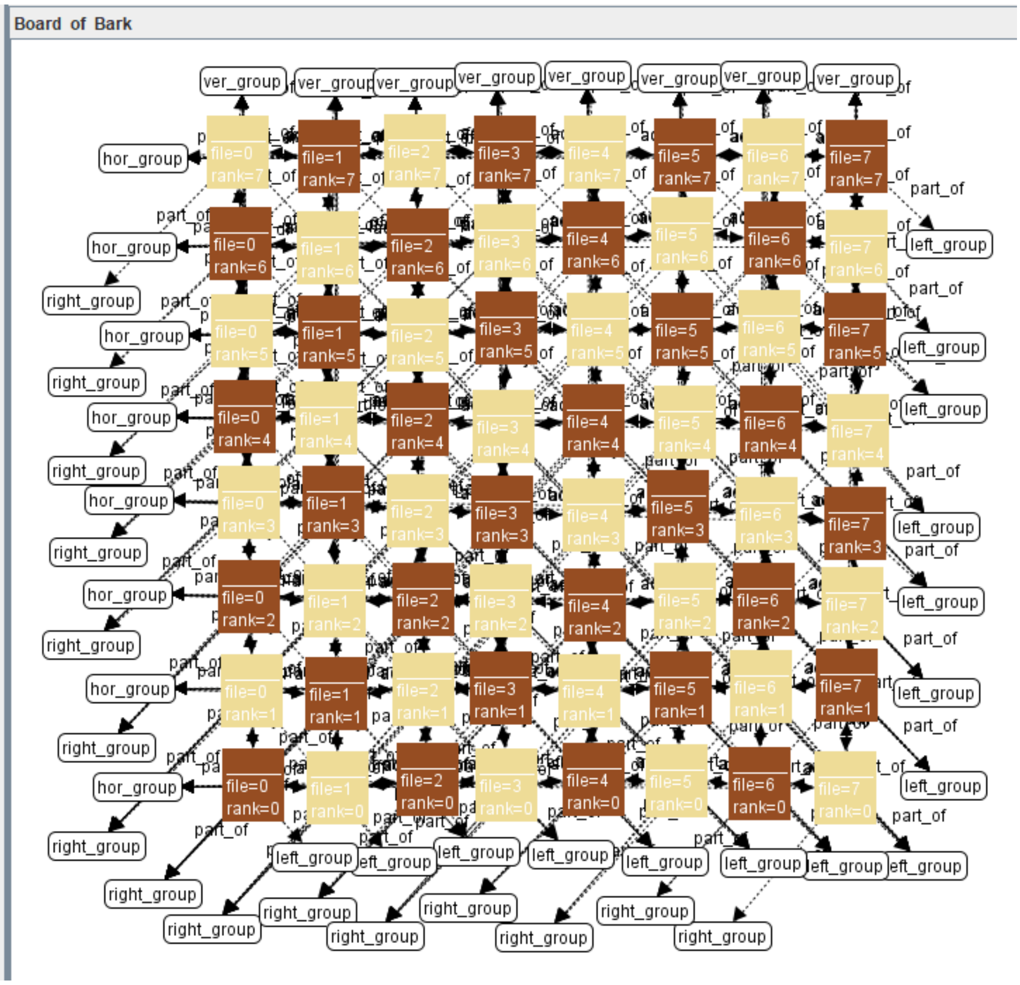

Figure 9: The board after hand-made ordering.

3 Representing game states and enabling complex behaviors

Pieces are represented as nodes connected (via an ***at***) to the squares they are on. With a boolean `color` attribute (`false` for black pieces and `true` for white ones), **piece** is the parent class, which is inherited by the **pawn**, **knight**, **bishoplike**, **rooklike**, and **king** nodes. Specifically, the **bishoplike** and **rooklike** nodes are further inherited by **bishop**, **rook**, and **queen** nodes, the latter of which is the only example of multiple inheritance in the system (as it moves like a rook and a bishop). A game state is not only composed by the pieces' positions on the board, but also by the information about the current rights to move, the castling rights, the last move, and the move counter (ply). This information is stored, explicitly via attributes and implicitly via edges, in the **game_state**.

Not only structure Until now, we've only focused on defining a representation for the game, without including a description for its behavior. Graph rewriting mechanisms by themselves are enough to define a description for the rules of a game such as chess. But, to simplify the design of complex algorithms, we define another set of nodes all inheriting from **guard**. These compose a stack (through the use of ***next_guard***) of operations which are used to enable only a subset of the rules at a time. Thanks to these, we were able to define the generation of moves, the evaluation of the board, the exploration of the game tree, and more.

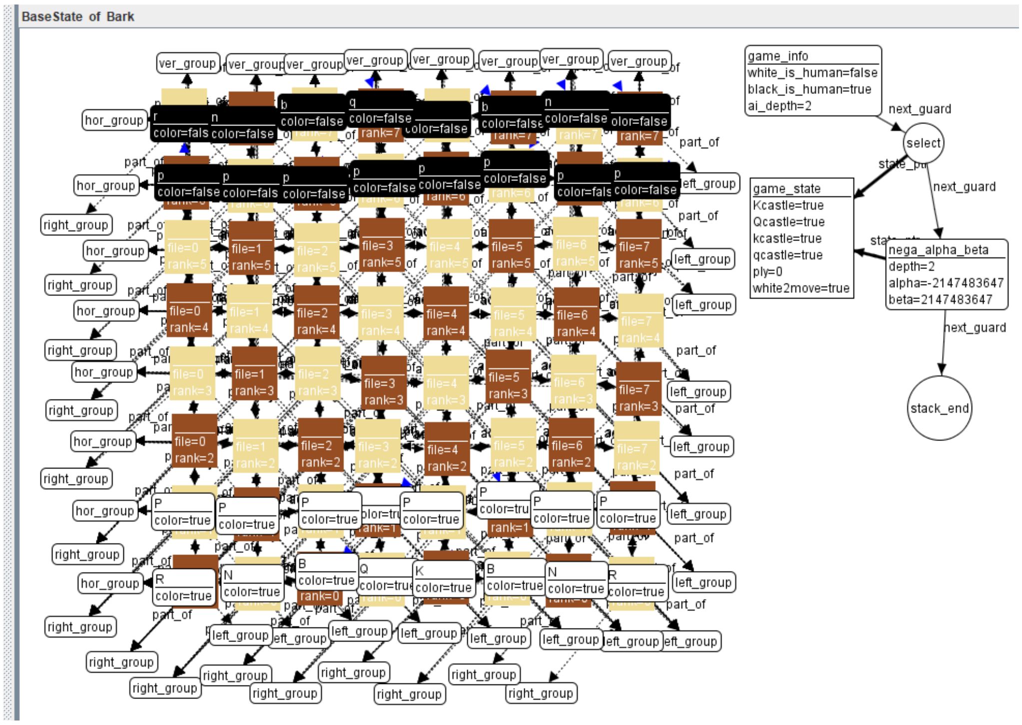

Figure 10: The starting configuration of the board.

Out of these **guard** nodes, there are three that are designed for the control of the stack itself:

1. **game_info** acts both as the start of the stack and as a container for the game information (which player is human and the depth of the search).
2. **void_guard** has a usage akin to a "no-op" operation, as it is used to block all rules until it is removed.
3. **stack_end** is the end of the stack, and it is used to signal which guard is at the top of the stack.

All guards can have a ***state_ptr*** which connects them to a **game_state**.

4 Generating moves

How it is done As cited previously, in a traditional chess engine a lot of computations are done via "magic bitboards". These are perfect hashing algorithms which can trivialize a lot of checks, even the generation of movements. Without getting into the details, the moves are produced in a two-phase manner:

1. The result of every possible move is generated (via magic bitboards), without any forward checking (e.g. leaving the king in check), as that is usually too complex and slow to implement. The resulting set is called "pseudo-legal moves".
2. For every possible resulting board, the king is checked for being in check. If it is, the move is discarded. This filtered set is called "legal moves".

A further information to keep into account is that a chess engine usually enables "push" and "pop" operation on moves (i.e. to make a move and then undo it). This is done by keeping a stack of the boards, and pushing a new configuration every time a move is made. This is a very efficient way to implement the "undo" functionality, as it doesn't require to recompute the whole board every time. Furthermore, it is also used to implement the exploration of the game tree, as it allows backtracking to the previous state of the game.

How Bark does it Bark, of course, cannot use magic bitboards. To produce a move, we have the **produce** guard, which is "checked" upon by all the `move_*` and `capture_*` rules. Our method to generate moves is akin to the traditional one, but since we can afford to hold only a single board instance, it is more complex (and slow). We represent the game tree by connecting the **game_state** nodes: when pushing and popping the moves, instead of retrieving a stored configuration, we use the information present on the **game_state** to execute or revert the move by operating on the board instance. To implement push, pop and threat check operations we design multiple rules and their respective guards (**push**, **pop**, and **threat_check**), which we'll describe in further sections.

To avoid repetitions, we first describe the patterns common to most (if not all) the `move_*` and `capture_*` rules:

- All rules produce a new **game_state** node, which is connected to the current one via a ***next_state*** edge. The moved piece gets a ***mover*** connecting it to the new state, and the new state gets connected to the arriving square through a ***to*** edge. All but a couple rules append **pop**, **threat_check**, and **push** (which we define in the next section) to the stack. For the sake of the **threat_check**, moves will require the king of the moving color, to pass its position to **threat_check**.

- All rules have a NAC `already_added` to ensure that the rule has not been applied yet.

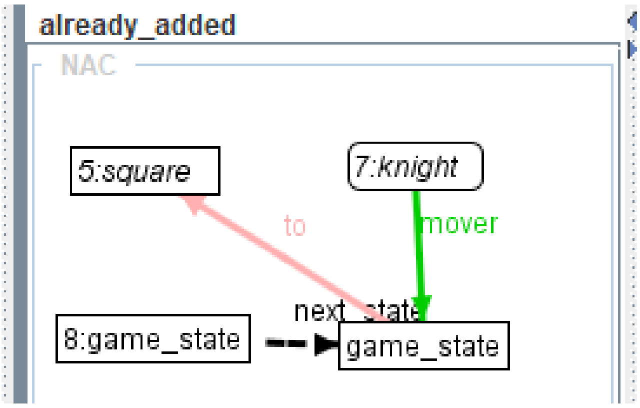

Figure 11: The `already_added` NAC of the `move_knight` rule.

The rules are:

1. `move_pawn` applies only when the next square is empty (`no_blocker` NAC) and the pawn is moving forward (`r2==r1+(c?1:-1)`). It also checks that the move will not lead to a promotion (`r1!=(c?6:1)`), as it is handled by another rule.

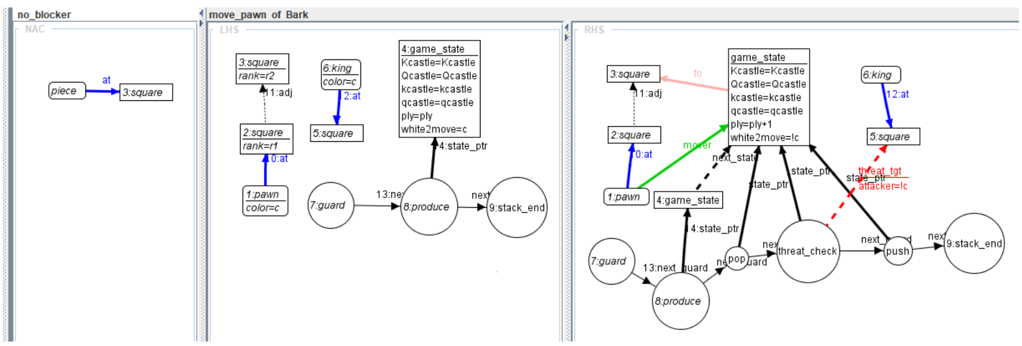

Figure 12: `move_pawn`

2. `move_pawn_promotion_white/black` apply like the `move_pawn` rule, but only when the move leads to a promotion (i.e. the rank reached is either 7 or 0, based on moving color). These are among the only

rules that don't append **pop**, **threat_check**, and **push** to the stack, but make a more complex update to it (as each possible promotion is a different move). Furthermore, each **game_state** is connected to the promotion piece via a ***promotion_into***edge.

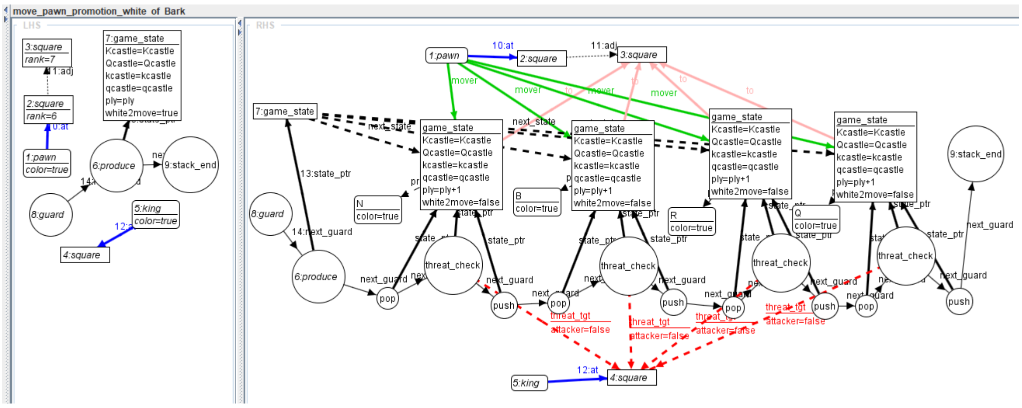

Figure 13: `move_pawn_promotion_white`

3. `move_pawn_double` applies only when both next squares are empty (`no_blocker` and `no_middle_blocker` NAC) and the pawn is moving forward (`r2==r1+(c?2:-2)`). It also checks that this is done only when it is the first move of the piece (`r1==(c?1:6)`).

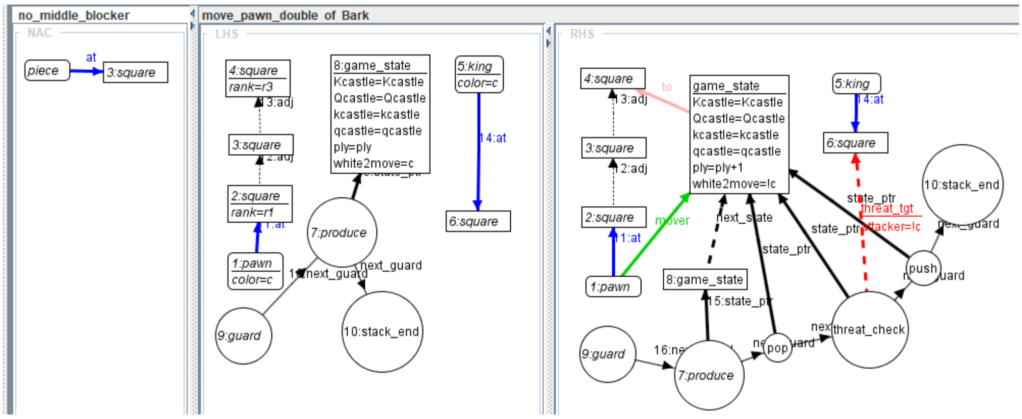

Figure 14: `move_pawn_double`

4. `capture_pawn` applies only when the next square contains a foe or an en passant is possible (`normal_capture | en_passant` GAC) and the pawn is moving forward (`r2==r1+(c?1:-1)`). It also checks that the move will not lead to a promotion (`r1!=(c?6:1)`), as it is handled by another rule.

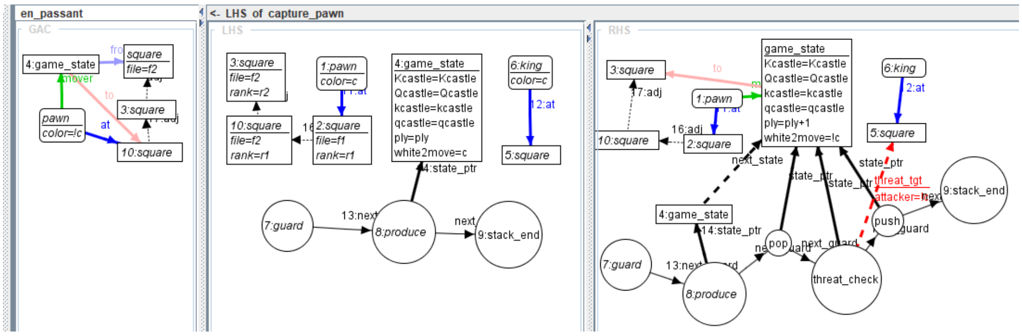

Figure 15: `capture_pawn`

5. `capture_pawn_promotion_white/black` apply like the `capture_pawn` rule, but only when the move leads to a promotion (i.e. the rank reached is either 7 or 0, based on moving color). Again, these are among the only rules that make a more complex update to the stack (as each possible promotion is a different move). Unlike the normal `capture_pawn` rule, there is no need to check for en passant, as it is impossible when a promotion is happening. Again, each **game_state** is connected to the promotion piece via a ***promotion_into***edge.

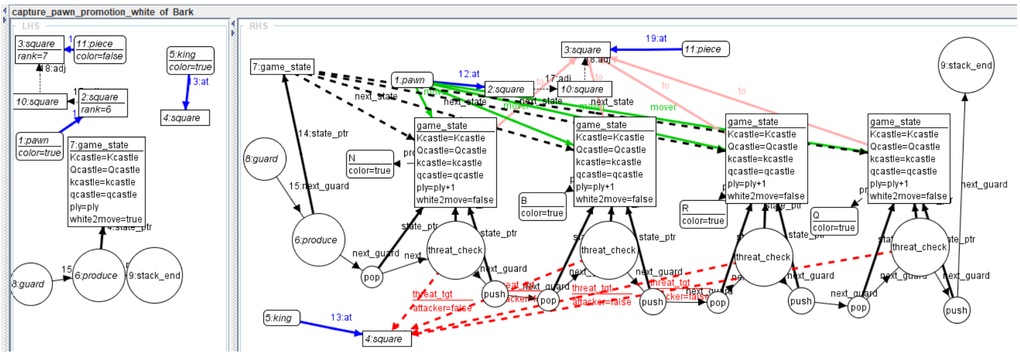

Figure 16: `capture_pawn_promotion_white`

6. `move_knight` applies both to normal moves and captures (indeed, there doesn't exist a `capture_knight`). To ensure that the piece can only move to an empty square or over a foe, a `no_blocker` NAC is added.

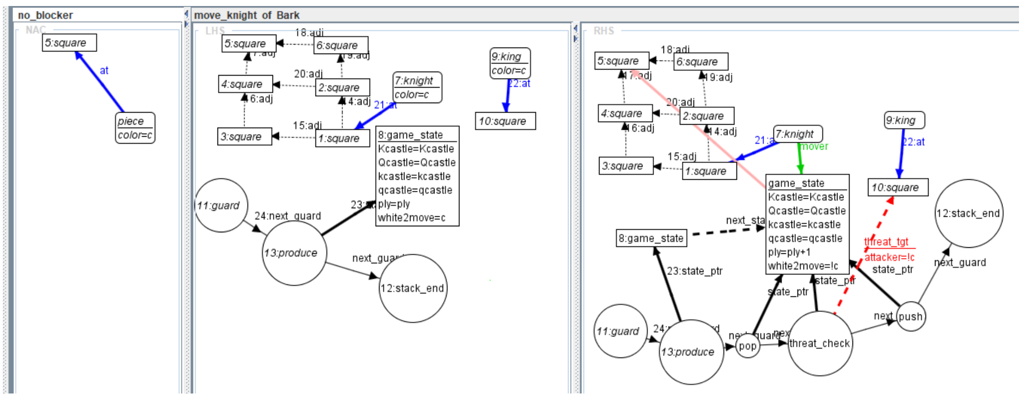

Figure 17: `move_knight`

7. `move_bishop_like` applies both to normal moves and captures, to both bishops and queens. To ensure that the piece can only move to an empty square or over a foe, a `no_blocker` NAC is added. Furthermore,

to ensure that there is no piece in between the starting and arriving squares, a `no_middle_blocker` NAC is added; this is possible only when the piece is in the same **bishop_like_group**, and it abides

$$(8(f_3 - f_1) + r_3 - r_1)(8(f_3 - f_2) + r_3 - r_2) < 0 \tag{1}$$

which is the condition for the middle (f_3, r_3) square to be between the starting (f_1, r_1) and arriving (f_2, r_2) squares.

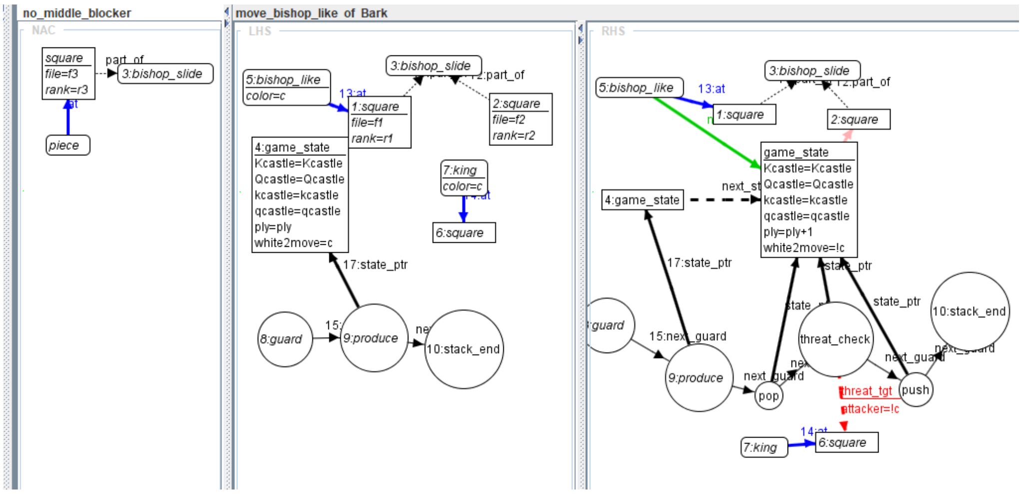

Figure 18: `move_bishop_like`

8. `move_rook` applies both to normal moves and captures, but only to rooks (as these have to disable castling when it is the first time they move, queens do not). It works similarly to the `move_bishop_like` rule, with the only difference that the common slide group must be a **rook_like_group**. Again, the possible middle blocker at square (f_3, r_3) must abide 1.

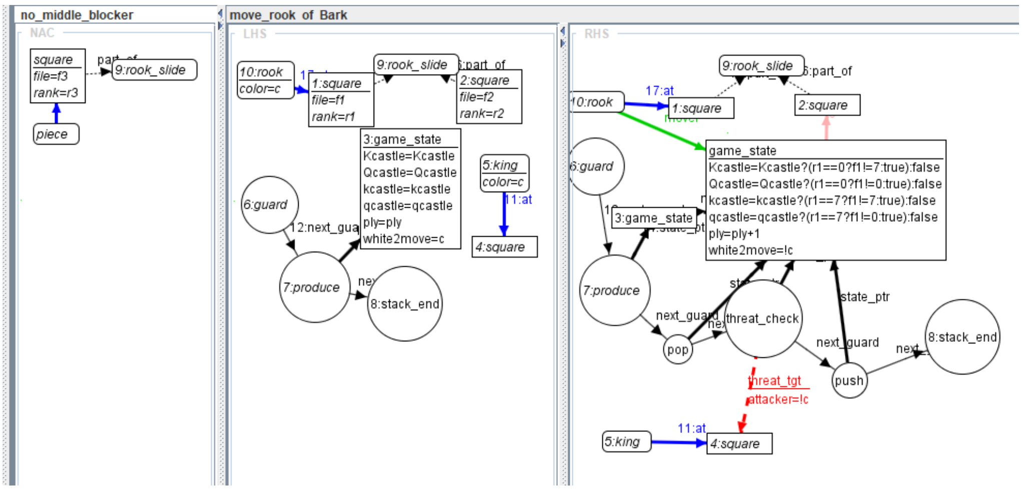

Figure 19: `move_rook`

9. `move_queen` acts equivalently to `move_rook`, with the only exception that it doesn't disable castling.

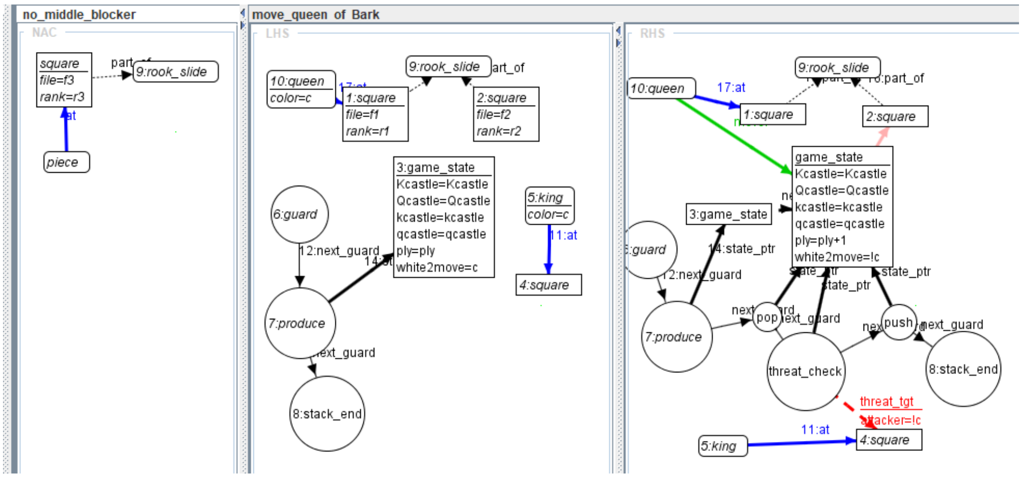

Figure 20: `move_queen`

10. `move_king_adj` applies both to normal moves and captures. To ensure that the piece can only move to an empty square or over a foe, a `no_blocker` NAC is added. Of course, as it is the king itself that is moving, the **threat_check** checks on the arriving square, also, as the king is moving, the castling rights are updated.

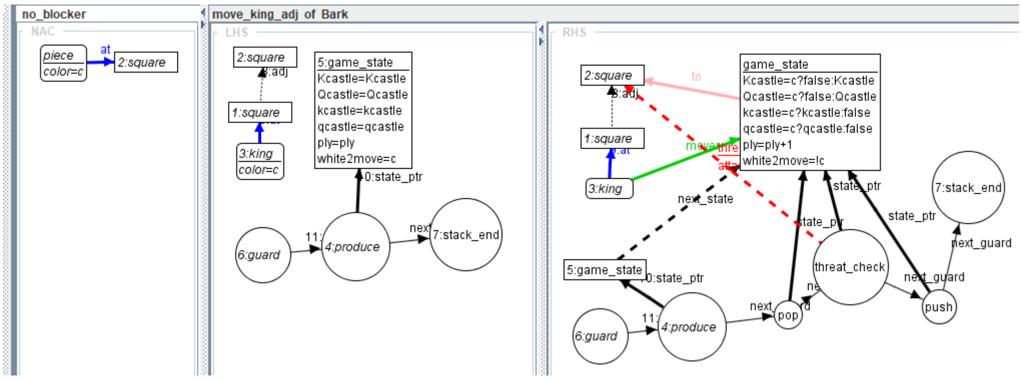

Figure 21: `move_king_adj`

11. `move_king_diag` acts the same as `move_king_adj`, the only difference is in the diagonal moves.

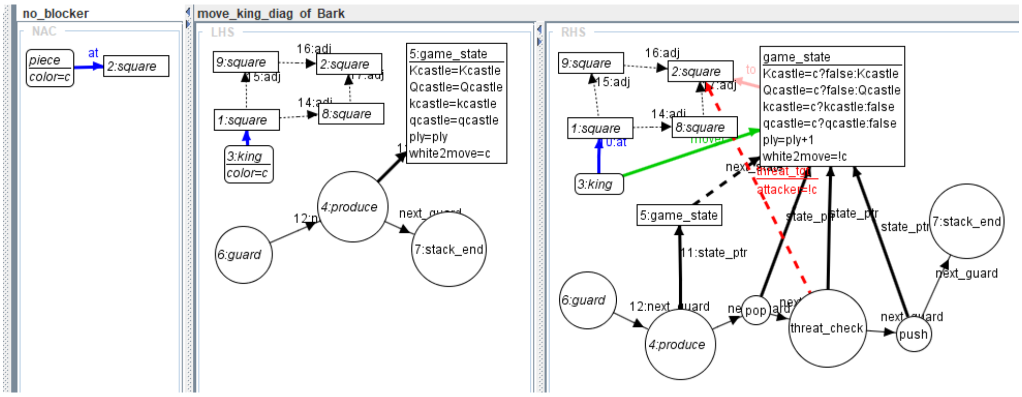

Figure 22: `move_king_diag`

12. `move_kingside/queenside_castle` execute castling (if it is enabled in the current **game_state**, `c?Kcastle:kcastle` and `c?Qcastle:qcastle`). They also check that the squares between the king and the rook are empty (`no_blocker*` NACs). As castling can only be done when the king is not in check (or moving across checked squares), these are the only moves that append **threat_check** to the stack before **push**. Of course the resulting game state has the castling rights updated.

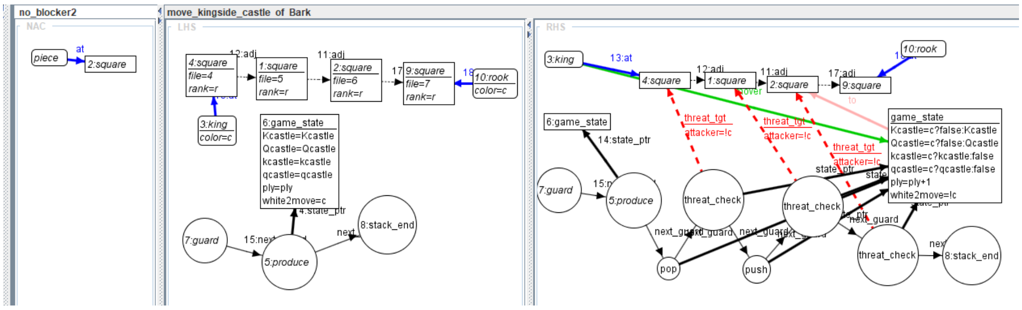

Figure 23: `move_kingside_castle`

5 Threat check, push, and pop

As mentioned, to discern between a legal move and a non-legal one, but also to enable exploration, we devised three guards: **push**, **pop**, and **threat_check**. Along with these guards, we have multiple rules accompanying them, which we describe.

Threat check All the `move_*` and `capture_*` rules append one or multiple **threat_check**s to the stack. A single instance of this guard is used to check if a square, connected via the ***threat_tgt*** edge, is under threat by a given color, defined by the `attacker` attribute. This check is only used to discard moves that would leave the king in check or otherwise make the rule illegal, therefore we don't model it for cases such as en passant (for which the king can't be affected). If any rule applies, the **game_state** is flagged via the ***illegal*** edge; but we do not remove it from the graph immediately, as it is needed to book-keep which pseudo-legal moves have been checked. To do this, we have the following rules:

1. `threat_pawn` applies if `r2==r1+(c?1:-1)`.

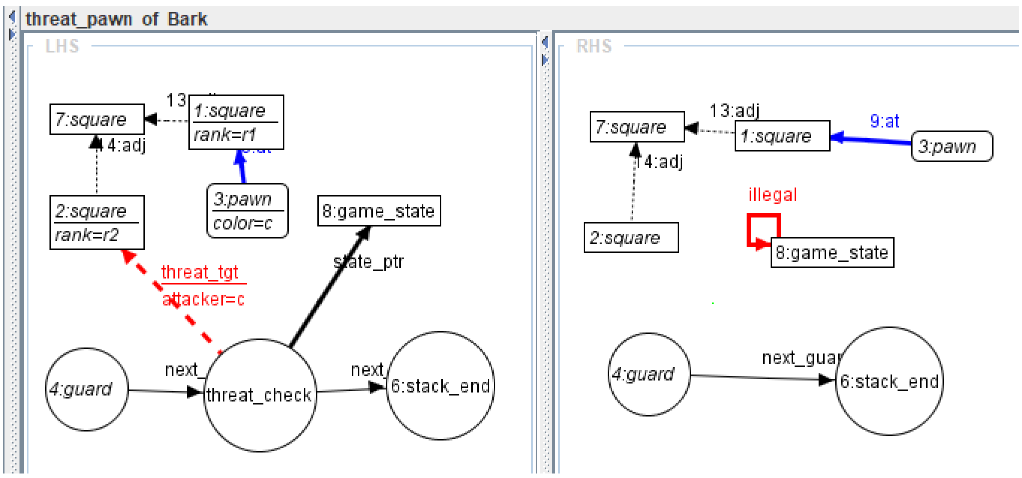

Figure 24: `threat_pawn`

2. `threat_knight` is one of the simplest rules, it doesn't have any NACs!

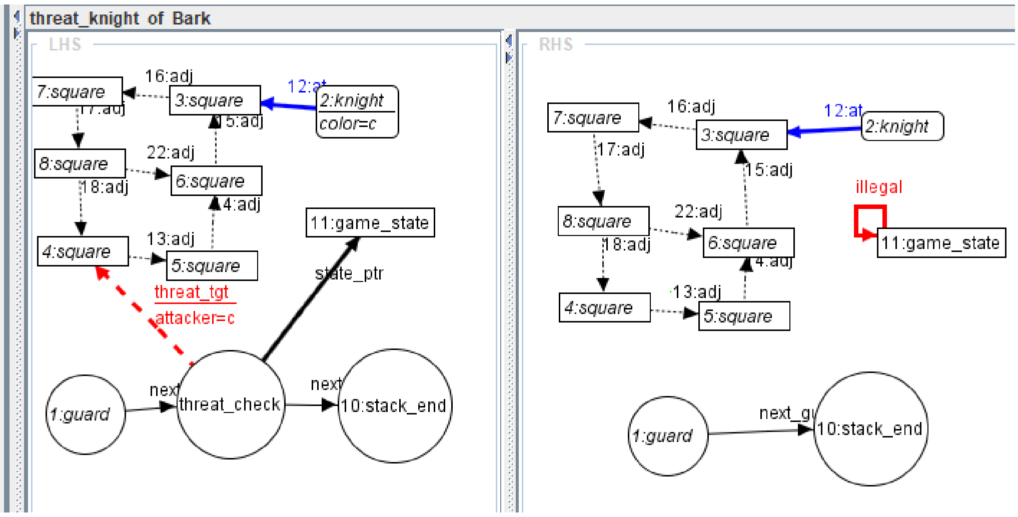

Figure 25: `threat_knight`

3. `threat_bishop/rook_like` are equal rules, with the only exception that one checks for attacks on diagonals and the other on vertical/horizontal lines. Both check the presence of a possible middle blocker via the `no_blockers` rule, which again follow 1.

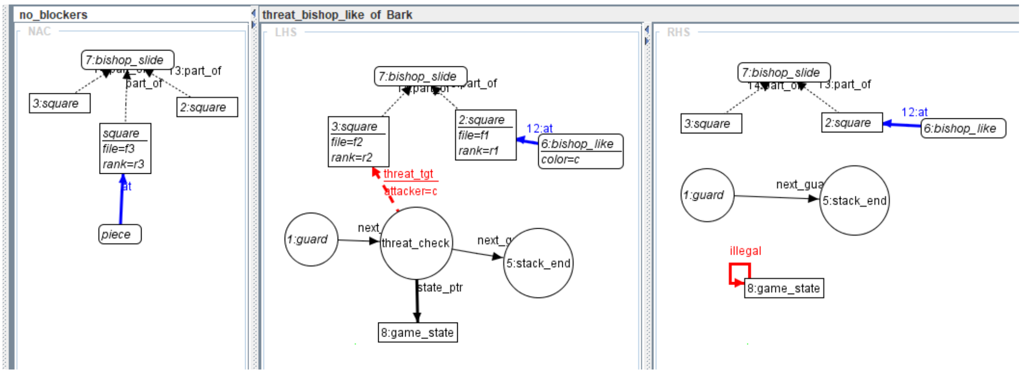

Figure 26: `threat_bishop_like`

4. `threat_king` applies for attacks from a king both from adjacent squares and diagonals, using a GAC which checks `(adj | diag)`.

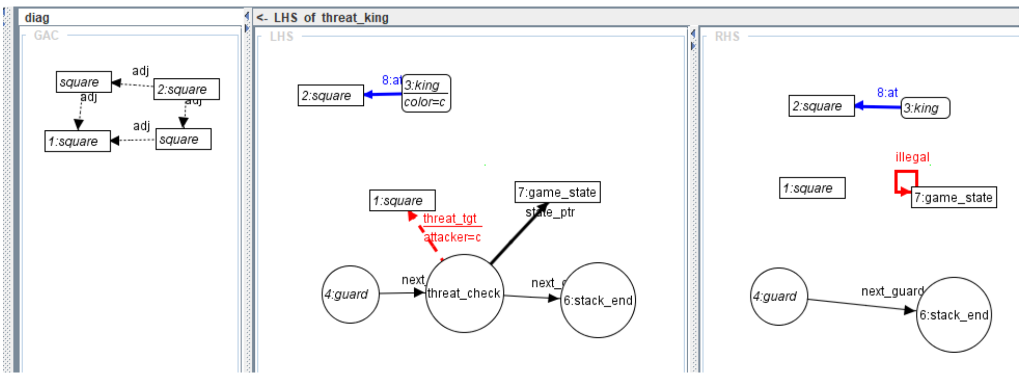

Figure 27: `threat_king`

Push All the push_* rules apply have the common behavior of adding, to the pushed **game_state**, a ***from*** edge pointing to the square the moving piece is coming from. This is used to enable the pop_* rules to revert the move. The rules are:

1. push_move is the most generic of moves, and indeed it is the most "restricted" of the bunch, as it may apply to the result of any move_* or capture_* rule. To restrict it, we have is_capture, is_promotion, is_en_passant and is_castling NACs. We have to devise a specific is_en_passant NAC as all other captures can easily be checked by whether there is an enemy piece on the arriving square, but en passant requires the presence of a foe pawn on the same file as the moving pawn.

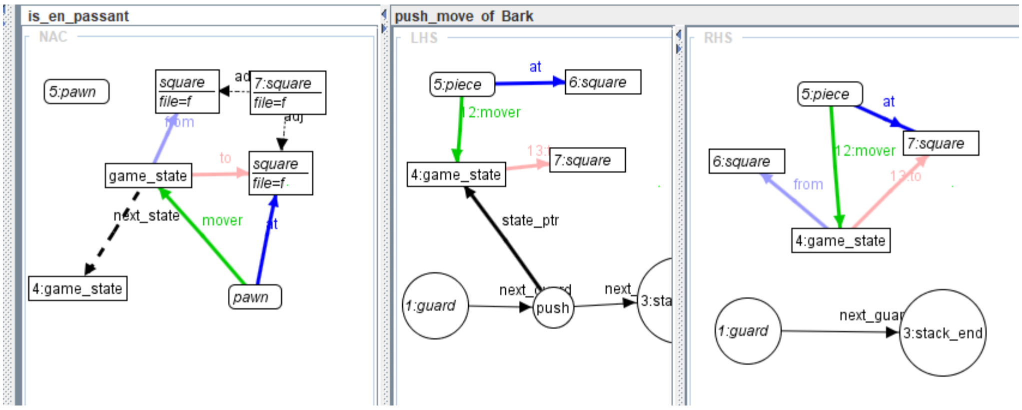

Figure 28: push_move

2. push_capture applies whenever the **game_state** points the ***to*** edge to a square with an enemy piece. Its only NAC is is_promotion, as there is a specific rule for that.

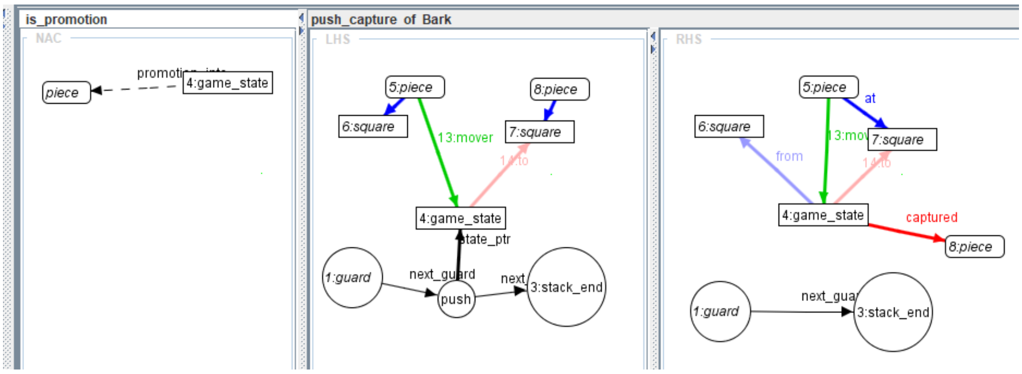

Figure 29: push_capture

3. push_en_passant applies to a very specific pattern, as it is the only capture that doesn't require the presence of an enemy piece on the arriving square.

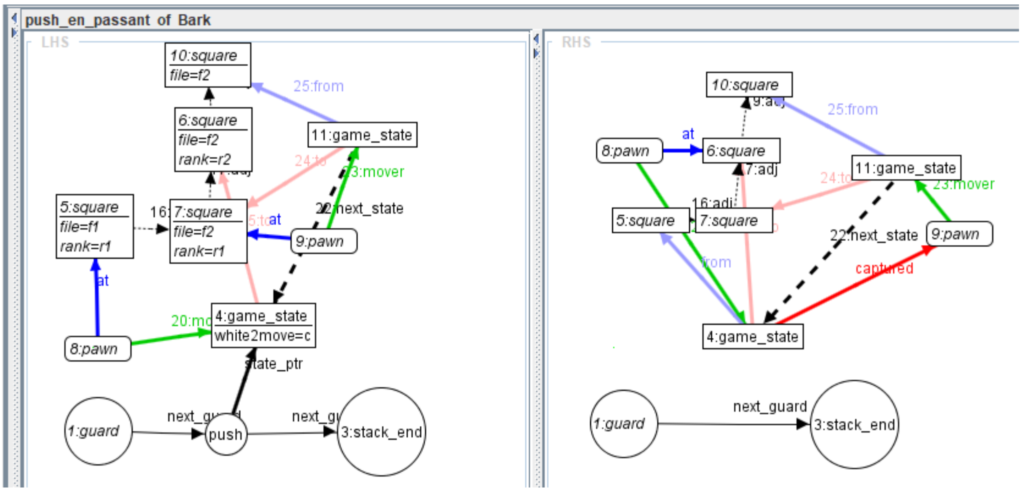

Figure 30: `push_en_passant`

4. `push_castling` too is very specific, it is the only move where the king moves two squares and two pieces are moved at once. It also has to abide `f3==2?f2==3&&f1==0:f2==5&&f1==7`, which checks that the rook is in the right position.

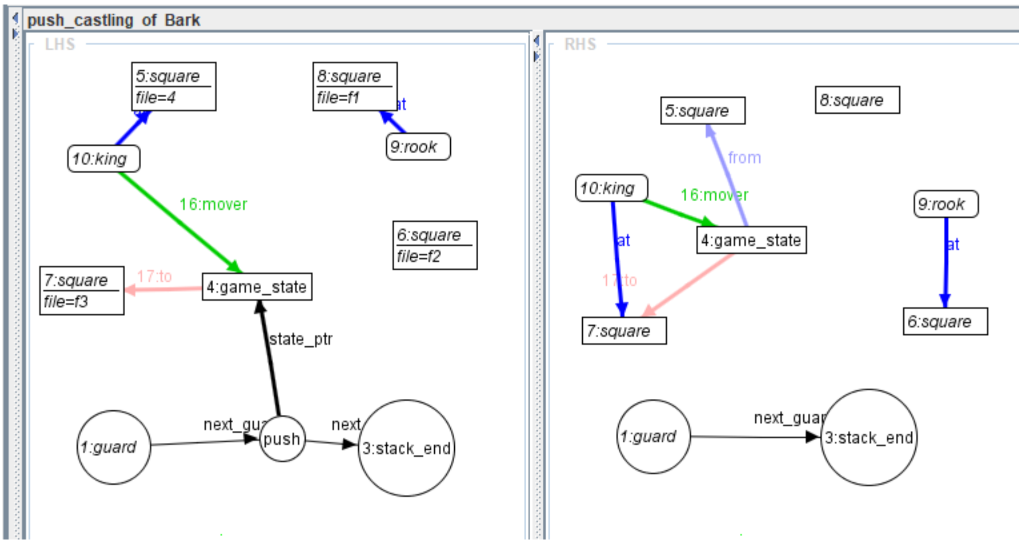

Figure 31: `push_castling`

5. `push_promotion` applies only when the **game_state** points to a promotion piece via the ***promotion_into*** edge. In this case we need a NAC to make sure that the rule is not a capture, as there is a specific rule for that.

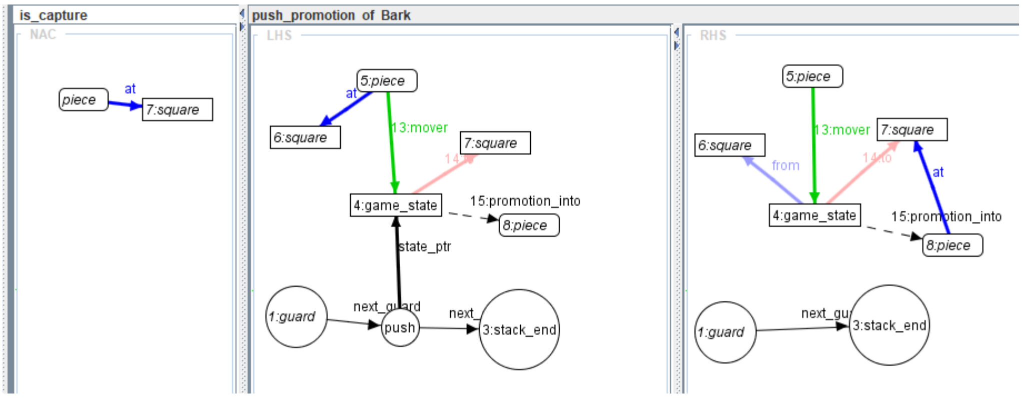

Figure 32: `push_promotion`

6. `push_promotion_cap` works in much the same way of `push_promotion`, with the only difference that it is a capture.

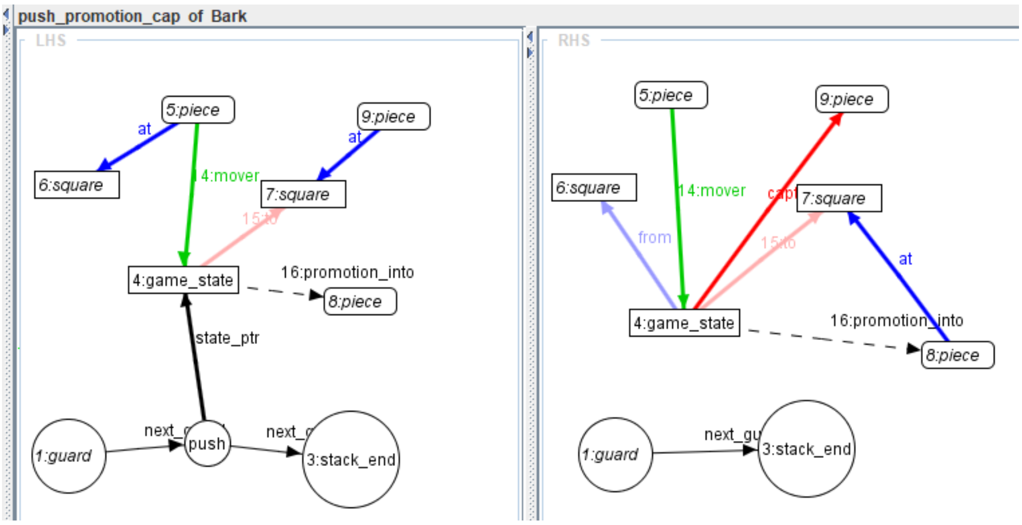

Figure 33: `push_promotion_cap`

Pop As one might assume, the `pop_*` rules are the inverse of the `push_*` rules, and can be found to be quite similar to their counterparts. Thanks to the ***from*** edges, these can be executed. They are:

1. `pop_move` it has `is_capture` and `is_castling` NACs but, unlike its counterpart `push_move`, it doesn't need to check for promotions, as a promoted rule doesn't have the ***mover*** edge; nor does it need to check for en passant, as en passants are handled by `is_capture` (in this case).

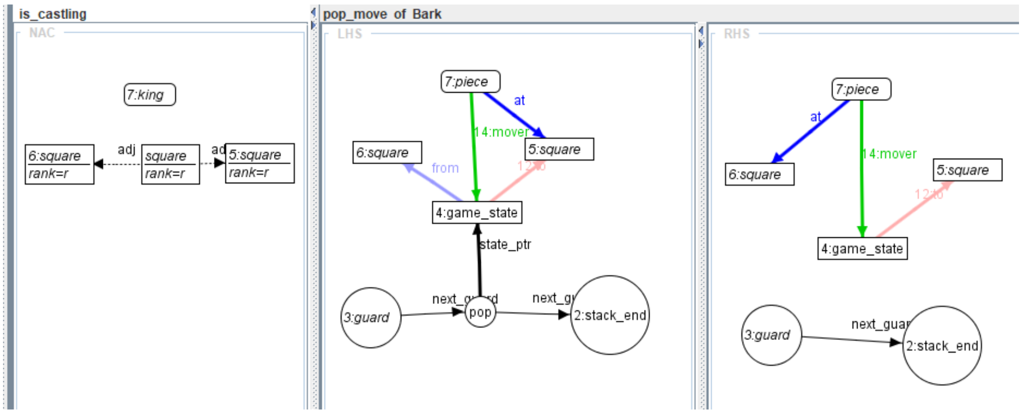

Figure 34: `pop_move`

2. `pop_capture` again, this doesn't have to check for promotions, as they don't have the ***mover*** edge. But it has to check for en passants via the `is_en_passant` NAC, as, again, they are handled by another rule (as the captured piece cannot be put in the square pointed by ***to***).

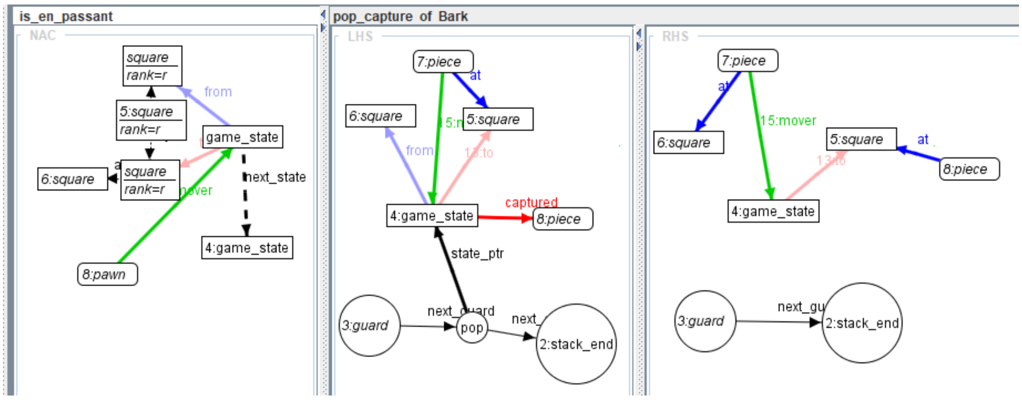

Figure 35: `pop_capture`

3. `pop_en_passant` like its counterpart, is quite specific (therefore the lack of NACs).

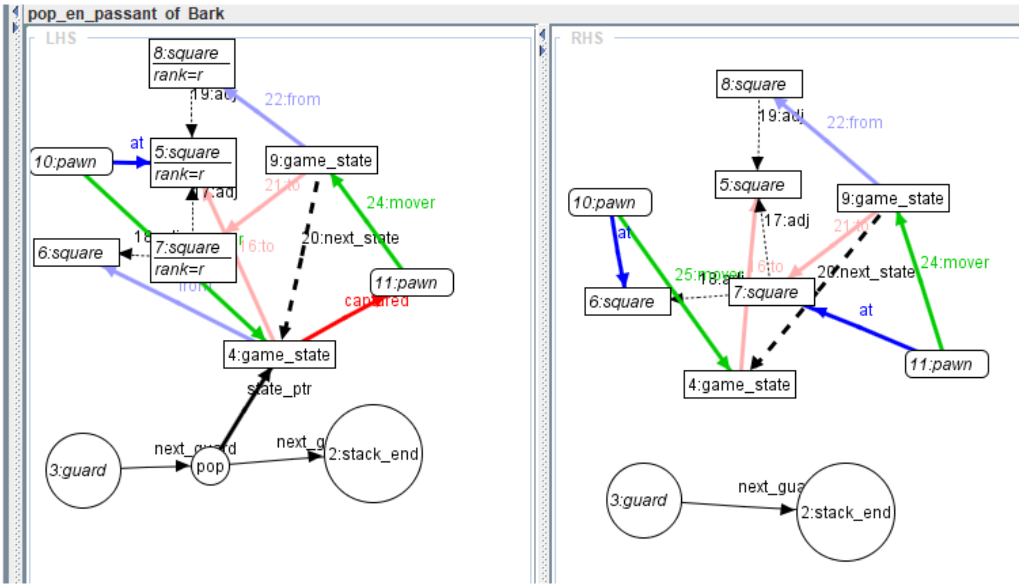

Figure 36: `pop_en_passant`

4. `pop_castling` too, like its counterpart, has to abide a rule to check that the rook is in the right position `f1==2?f2==3&&f3==0:f2==5&&f3==7`.

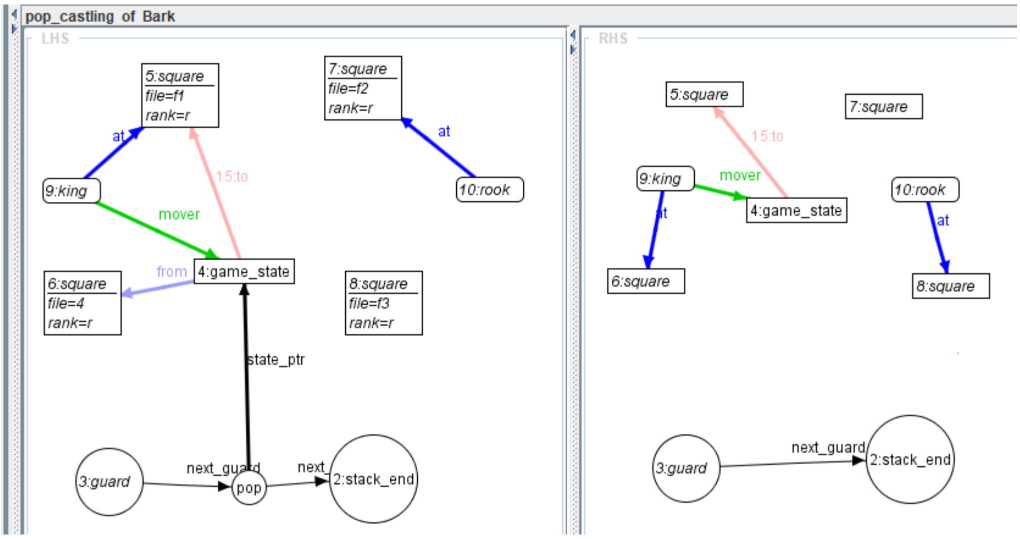

Figure 37: `pop_castling`

5. `pop_promotion` checks that the rule is not a capture, as it is handled by another rule (again).

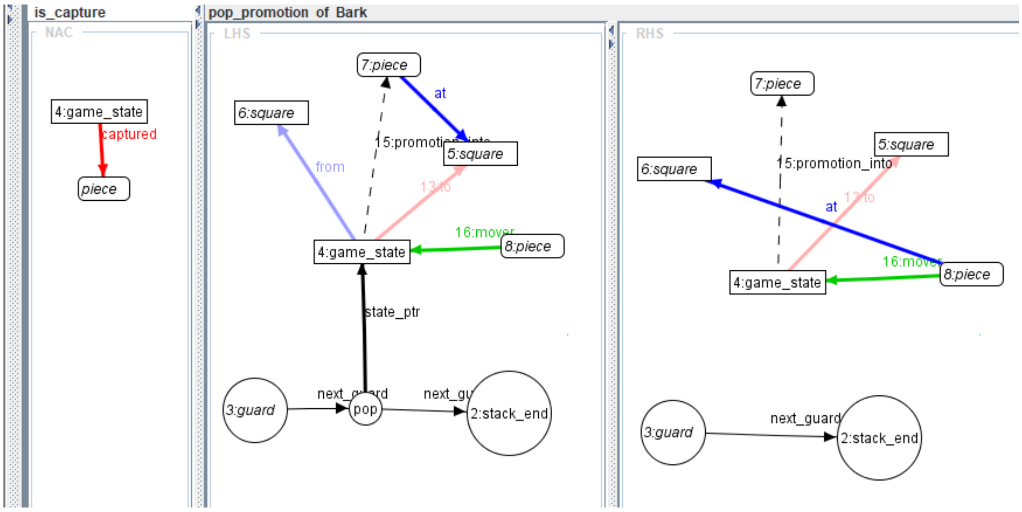

Figure 38: `pop_promotion`

6. `pop_promotion_cap` is the same as `pop_promotion`, but for captures.

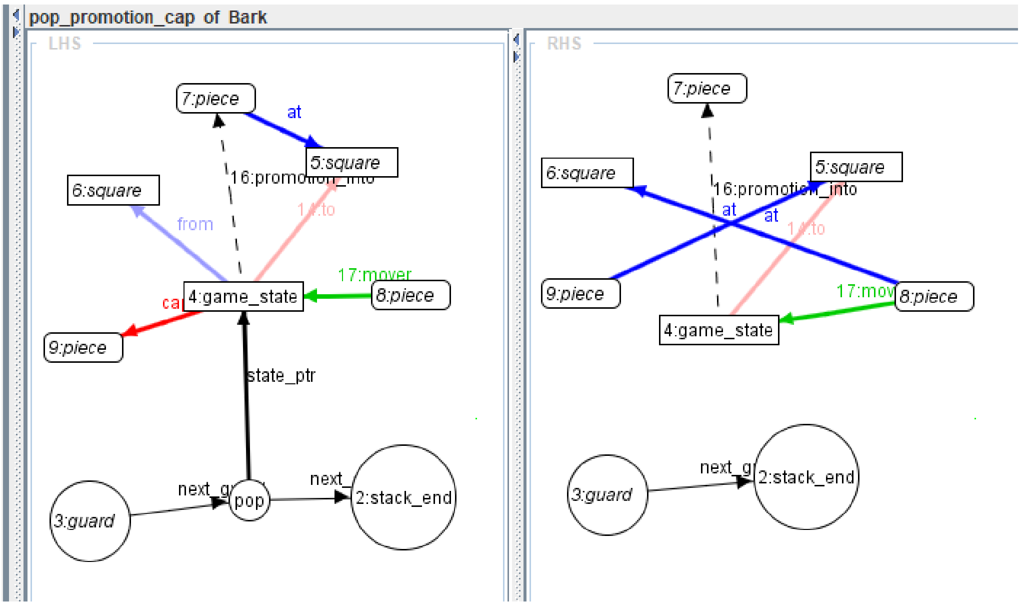

Figure 39: `pop_promotion_cap`

6 Evaluation of the board

How it is done In traditional chess engines, the evaluation of the board is a complex task, as it requires to take into account a lot of factors, such as the position of the pieces, the pawn structure, the king safety, and more. This is a very complex task, as it requires a lot of heuristics and domain knowledge.

How Bark does it To save computing time, and to make it as simple as possible, in Bark we take into account three factors:

- The material difference between the two players.
- The position of the pieces on the board.

- The game phase (i.e. the number of pieces on the board).

We do this by following the PeSTO (Piece Square Tables Optimized) approach, which is a simple heuristic that assigns a value to each piece based on its position on the board and the game phase, which are very important factors in chess. To this end, we have defined six `eval_*` rules, which are applied when the **evaluate** guard is present. Except for the piece types, the rules are all the same, and they follow the structure:

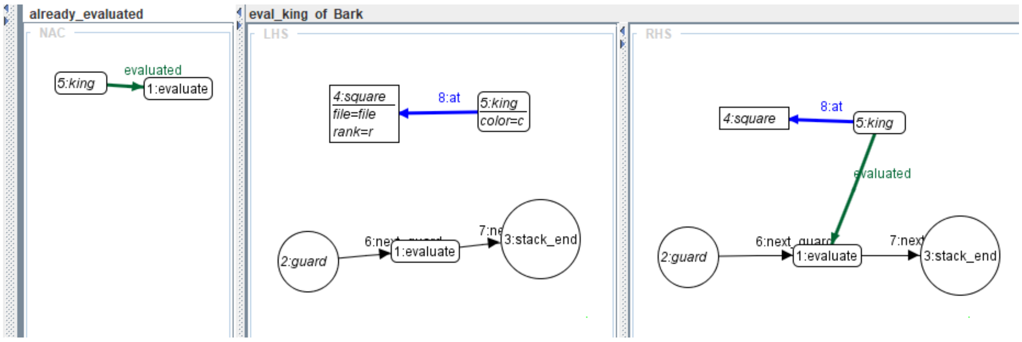

Figure 40: `eval_king`

The real difference between the rules is in how they change the attributes of the **evaluate**. The original PeSTO evaluation works by using a set of tables, which are used to assign a value to each piece based on its position on the board and the game phase. In Bark, since we can't create arrays, we have rewritten these tables as a simple (and very unwieldy) set of nested ternary operators. As doing such a thing by hand would've been tedious and error-prone, we have written a formatter in Python that takes the values of the tables and outputs the nested ternary operators.

The resulting evaluation enables us to avoid the need for more complex rules and further computation, which would have been needed to create the value of the board.