Exercises

Computer Aided Verification

Author René Schwaiger
Mail sanssecours@f-m.fm
Version 2
Date September 18, 2013

Contents

1 · Binary Decision Diagrams

1.1 Exercise 1

Give a linear time algorithm for BDD isomorphism as defined on page 9.

1.1.1 Solution

```
#!/usr/bin/env python
# coding=utf-8

# ----------------------------------------------------------------------------
#   Various code to create and operate on binary decision diagrams.
#
#   Version: 2
#   Date:    2013-07-12
#   Author:  René Schwaiger (sanssecours@f-m.fm)
# ----------------------------------------------------------------------------

# -- Classes -----------------------------------------------------------------


class Terminal(object):
    """A terminal node of a BDD."""

    def __init__(self, value=True):
        """Initialize a new terminal node.

        Arguments:

            value – The (binary) value of the terminal node (default: True)

        """
        self.value = value

    def __eq__(self, bdd_node):
        """Compare this terminal to an other BDD node.

        Arguments:

            bdd_node – The BDD node to compare this terminal against.
```

```
        Examples:

            >>> terminal_true = Terminal()
            >>> terminal_true == Terminal(True)
            True
            >>> terminal_true == Terminal(False)
            False

        """
        try:
            # Compare two terminals
            return self.value == bdd_node.value
        except AttributeError:
            # A terminal can never be equal to a non-terminal object
            return False


class Node(object):
    """A node of a binary decision diagram."""

    def __init__(self, low, high, variable):
        """Initialize a new BDD node.

        Arguments:

            low      - The low child of the node.
            high     - The high child of the node.
            variable - The variable represented by this node.

        """
        self.low = low
        self.high = high
        self.variable = variable

    def __eq__(self, node):
        """Compare this BBD nodes to an other BDD node.

        Arguments:

            node - The BDD node to compare this node against.
```

```
        Examples:

            >>> terminal_false = Terminal(False)
            >>> terminal_true = Terminal(True)
            >>> node = Node(terminal_false, terminal_false, 'p')
            >>> node2 = Node(node, terminal_true, 'p')
            >>> node == node2
            False
            >>> node2 == Node(node, terminal_true, 'p')
            True

        """
        try:
            return (self.variable == node.variable and
                    self.low == node.low and
                    self.high == node.high)
        except AttributeError:
            return False


class BDD(object):
    """Saves data contained in a binary decision diagram."""

    def __init__(self, root):
        """Create a new BDD.

        Arguments:

            root - The root of the new BDD. This can be either a "normal" node
                   or a terminal node.

        Examples:

            >>> terminal_true = Terminal(True)
            >>> bdd = BDD(terminal_true)
            >>> BDD = Node(terminal_true, terminal_true, 'a')

        """
        self.root = root
```

```
    def isomorphic(self, bdd):
        """Check if two BDDs are isomorphic or not.

        Both BDDs for this function need to be fully reduced and have the same
        ordering. Otherwise this function will return ``False`` although the
        BDDs might in fact be isomorphic.

        Arguments:

            bdd - The BDD to compare this BDD against.

        Examples:

            >>> terminal_false = Terminal(False)
            >>> terminal_true = Terminal(True)
            >>> BDD(terminal_true).isomorphic(BDD(terminal_false))
            False
            >>> node_b1 = Node(terminal_false, terminal_false, 'b')
            >>> node_b2 = Node(terminal_false, terminal_true, 'b')
            >>> node_a1 = Node(node_b1, node_b2, 'a')
            >>> BDD(node_a1).isomorphic(BDD(terminal_false))
            False
            >>> BDD(node_a1).isomorphic(BDD(node_a1))
            True
            >>> node_a2 = Node(node_b1,
            ...                Node(terminal_true, terminal_true, 'b'),
            ...                'a')
            >>> BDD(node_a1).isomorphic(BDD(node_a2))
            False

        """
        return self.root == bdd.root


if __name__ == '__main__':
    # Import and run doc-tests
    import doctest
    doctest.testmod()
```

1.2 Exercise 2

Describe a size-efficient BDD for the relation "a ≥ b" for n-bit integer numbers.

1.2.1 Solution

We already know that related variables should be close together in the ordering. Therefore we place the bits of the same significance after each other starting from the most significant bit of a. This leaves us with the ROBDD shown in Figure 1.

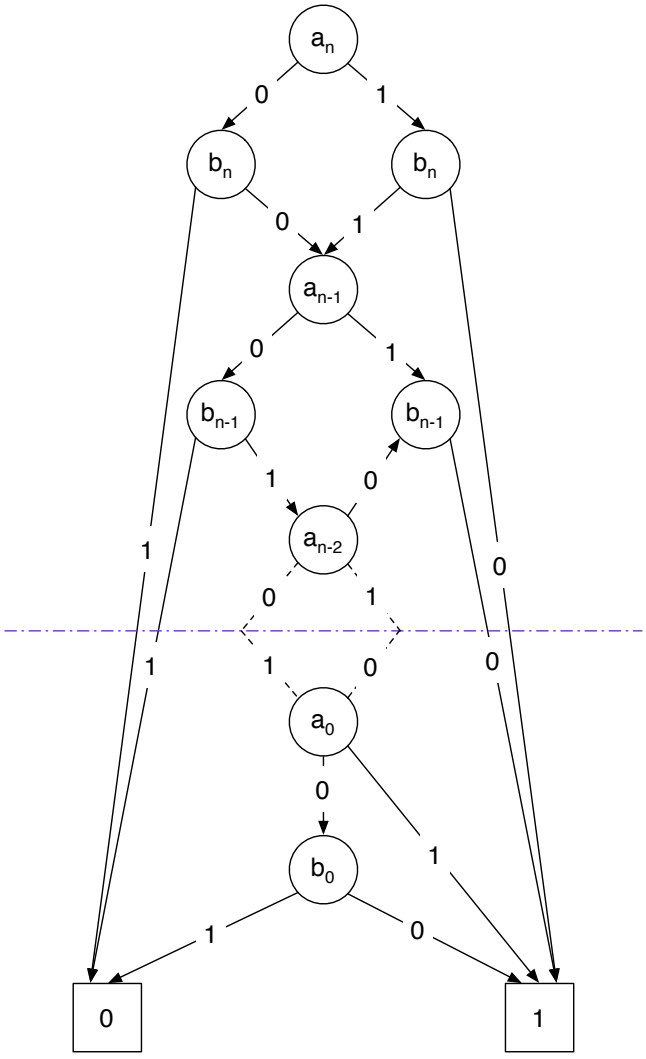


Figure 1: OBDD for the relation "a ≥ b"

1.3 Exercise 3

Describe an algorithm which transforms a BDD into an equivalent boolean formula.

1.3.1 Solution

The following code shows an algorithm which creates a boolean formula in disjunctive normal form. The basic idea behind the code is to create a conjunction of every variable visited on a path to a terminal node with value "1". We create the whole DNF-formula by joining the formulas for these paths by disjunction.

```
#!/usr/bin/env python3

# ------------------------------------------------------------------------------
#   Support for generating formulas from a given BDD.
#
#   Version: 1
#   Date:    2013-07-13
#   Author:  René Schwaiger (sanssecours@f-m.fm)
# ------------------------------------------------------------------------------

# -- Imports -------------------------------------------------------------------

from bdd import Node, Terminal  # noqa

# -- Functions -----------------------------------------------------------------


def bdd_to_formula(node, previous_variables=None):
    """Convert a binary decision diagram to a boolean formula.

    This function creates a formula in disjunctive normal form. For every path
    to the terminal "1", it creates a subformula, containing the variables of
    this path.

    previous_variables - A list containing the variables of the BDD nodes
                         already visited.

    Examples:
```

```
        >>> terminal_false = Terminal(False)
        >>> terminal_true = Terminal(True)
        >>> node_a2 = Node(terminal_false, terminal_true, 'a2')
        >>> node_b1_1 = Node(node_a2, terminal_false, 'b1')
        >>> node_b1_2 = Node(terminal_false, node_a2, 'b1')
        >>> node_a1 = Node(node_b1_1, node_b1_2, 'a1')
        >>> bdd_to_formula(node_a1)
        '(¬a1∧¬b1∧a2)∨(a1∧b1∧a2)'

    """
    # At the end of the path (terminal node)
    if isinstance(node, Terminal):
        return (['∧'.join(previous_variables)]
                if previous_variables and node.value
                else [str(node.value)])

    # We create a formula by joining the solutions from all paths by
    # disjunction
    if not previous_variables:
        formulas = bdd_to_formula(node.low, ['¬{}'.format(node.variable)])
        formulas.extend(bdd_to_formula(node.high,
                                       ['{}'.format(node.variable)]))
        # Filter empty terms (terminal node and therefore formula is false)
        formulas = [formula for formula in formulas
                    if not (formula.startswith(str(False)))]
        solution = '∨'.join(['({})'.format(formula) for formula in formulas])
        return solution

    # Extend the variables already visited with the current variable
    variables_low = previous_variables + ['¬{}'.format(node.variable)]
    variables_high = previous_variables + ['{}'.format(node.variable)]
    formulas = bdd_to_formula(node.low, variables_low)
    formulas.extend(bdd_to_formula(node.high, variables_high))
    return formulas

if __name__ == '__main__':
    # Import and run doc-tests
    import doctest
    doctest.testmod()
```

2 · Temporal Logic

2.1 Exercise 4

Prove the equivalence for $\mathbf{A}(f\,\mathbf{U}g)$ on page 16.

2.1.1 Solution

We need to prove that $\mathbf{A}(f\,\mathbf{U}g)$ is equivalent to $\neg E(\neg g\mathbf{U}\neg f \wedge \neg g) \wedge \neg\mathbf{EG}\neg g$. We use the following equivalences:

(1) $\mathbf{A}f \Leftrightarrow \neg\mathbf{E}\neg f$

(2) $\neg(f\,\mathbf{U}g) \Leftrightarrow (\mathbf{G}\neg g \vee \neg g\mathbf{U}\neg f \wedge \neg g)$ The formula above is true since for f until g to not hold:

   (a) g has to not hold at all (there exists no k such that $\pi^k \vDash g$) or

   (b) f might hold at first, while g does not hold, but g does not hold after that ($\neg g\mathbf{U}\neg f \wedge \neg g$).

(3) $\mathbf{E}(f \vee g) \Leftrightarrow \mathbf{E}f \vee \mathbf{E}g$

(4) $\neg(f \vee g) \Leftrightarrow \neg f \wedge \neg g$

to deduct the following proof:

$$
\begin{aligned}
\mathbf{A}\left(f\,\mathbf{U}g\right) &\overset{(1)}{\Leftrightarrow} \neg\mathbf{E}\neg\left(f\,\mathbf{U}g\right) \\
&\overset{(2)}{\Leftrightarrow} \neg\mathbf{E}\left(\left(\mathbf{G}\neg g\right) \vee \left(\neg g\mathbf{U}\neg f \wedge \neg g\right)\right) \\
&\overset{(3)}{\Leftrightarrow} \neg\left(\mathbf{EG}\neg g \vee \mathbf{E}\left(\neg g\mathbf{U}\neg f \wedge \neg g\right)\right) \\
&\overset{(4)}{\Leftrightarrow} \neg\mathbf{EG}\neg g \wedge \neg\mathbf{E}\left(\neg g\mathbf{U}\neg f \wedge \neg g\right)
\end{aligned}
$$

2.2 Exercise 5

Show the following lemma: Let M and N be two Kripke structures such that the transition relation of M is a superset of the transition relation of N. If an LTL property f holds on M, then f also holds on N.

2.2.1 Solution

Let us assume that a certain LTL formula $\mathbf{A}f$ holds on M. Since the transition relation of M is a superset of the transition relations of N, there is always the possibility to construct a Kripke structure equivalent to M by extending N with additional transitions/states. Since a LTL formula quantifies over ***all paths*** in a Kripke structure the set of all formulas which hold on N gets smaller when we extend N. This means that the set of all LTL formulas which hold on N is a superset of all the LTL formulas which hold on M. From this follows that, if a certain LTL formula $\mathbf{A}f$ holds on M, then this formula also has to hold on N.

2.3 Exercise 6

Show that $\mathbf{AFG}p$ is not logically equivalent to $\mathbf{AFAG}p$.

2.3.1 Solution

To show that $\mathbf{AFG}p$ is not equivalent to $\mathbf{AFAG}p$ we construct a Kripke structure which contains a state where $\mathbf{AFG}p$ holds but $\mathbf{AFAG}p$ does not. Figure 2 shows this Kripke structure ([VHZ11]), where $s_0 \vDash \mathbf{AFG}p$ holds, but $s_0 \vDash \mathbf{AFAG}p$ does not.

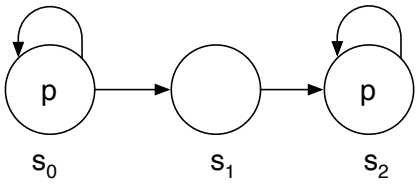


Figure 2: Kripe Structure, where $s_0 \vDash \mathbf{AFG}p$ holds, but $s_0 \vDash \mathbf{AFAG}p$ does not hold

$s_0 \vDash \mathbf{AF}(\mathbf{G}p)$ This formula specifies that in all paths sometimes in the future p will hold globally. This is true for the given Kripke structure since we start in s_0 in which p holds and either

- continue to stay in this state (p holds globally) or
- go to s_1 immediately followed by state s_2 (p holds globally).

$s_0 \vDash \mathbf{AF}(\mathbf{AG}p)$ The formula specifies that sometimes in the future for all paths $\mathbf{AG}p$ (for all paths p must always be true) has to

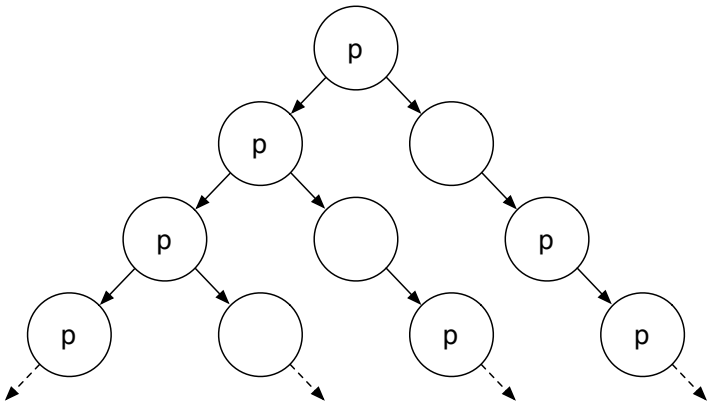


Figure 3: Computation Tree for Figure 2

hold. This is not the case if we follow the path on the left in the computation tree for the Kripke Structure (see Figure 3) since there is always a path on the right where p does not hold for every state.

2.4 Exercise 7

Describe a simple model checker for CTL over Kripke structures in pseudocode.

2.4.1 Solution

Clarke, Emerson and Sistla describe an algorithm to check CTL formulas in their article "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications" [CES86]. We use this algorithm to check if a CTL formula holds in a certain state of a Kripke structure.

All CTL formulas can be modeled, according to Clarke et al, using the following grammar:

$$f = p \mid \neg f \mid f \wedge f \mid \mathbf{AX}f \mid \mathbf{EX}f \mid \mathbf{A}[f\,\mathbf{U}\,f] \mid \mathbf{E}[f\,\mathbf{U}\,f]$$
$$p \text{ ... atomic proposition}$$

At the beginning we need to divide the given CTL formula f into its subformulas. A parser can be used for this purpose.

Each state of the Kripke structure is already labeled by the atomic propositions holding in it. Therefore we start with formulas of length 2 (e.g. $\mathbf{AX}p$, $p_1 \wedge p_2$), check in which state these formulas hold and label the states

accordingly. After that we continue with subformulas of length 3, then length 4 and so on, till we get to the length of the whole formula. After that we just need to check if a certain state s is labeled with the formula f to test if $s \vDash f$ holds.

```
# Partition the formula `f` into its subformulas
formulas = subformulas(f) ∪ f

# Go trough all subformulas beginning with the shortest formulas
for formula_length = 2…length(f):
    for formula in [formula for formula in formulas
                    if length(formula) == formula_length]:
         label(formula)

# Check if the CTL formula `f` holds in the given state
if f in formula(state):
    return True
else:
    return False
```

The function `label(formula)` needs to support all operands mentioned in the grammar of CTL. We start with the code for $\neg f$:

```
if formula.operator == '¬':
    for state in kripke_structure.states:
        # e.g. f = ¬g ⇒ f.first_subformula = g
        if not formula.first_subformula in state.labels:
            state.labels.add(formula)
```

The code for $f \wedge g$ looks quite similar:

```
if formula.operator == 'v':
    for state in kripke_structure.states:
        if formula.first_subformula in state.labels and
           formula.second_subformula in state.labels:
            state.labels.add(formula)
```

For formulas of the form $\mathbf{AX}f$ we check if f holds for every neighbor of a certain state.

```
if formula.operator == 'AX':
    for state in kripke_structure.states:
        # If the set of neighbours where the subformula does not hold is empty
```

```
        if not {neighbour for state.neighbours
                if formula.first_subformula in neighbour.labels}:
            state.labels.add(formula)
```

The operation needed to handle the operator $\mathbf{EX}$ are quite similar to the ones needed for formulas of the form $\mathbf{AX}f$. We need to check if f holds for one of the neighbours of each state.

```
if formula.operator == 'EX':
    for state in kripke_structure.states:
        # If the set of neighbours where the subformula holds is not empty
        if {neighbour for state.neighbours
            if formula.first_subformula() in neighbour.labels}:
                state.labels.add(formula)
```

For $A[fUg]$ we use the same methods as described by Clarke et al. We start with a depth first search on the states. For every unmarked state we call the procedure `au(state, formula)` .

```
if formula.operator == 'AU':
    # “Unmark” all states
    marked = {state: false for state in kripke_structure.states}
    for state in in kripke_structure.states:
        if not marked[state]:
            au(state, fomula)
```

In the procedure `au(state, formula)` we check if `formula` holds in the given state. We need to examine 2 basic cases where $\mathbf{A}[f\mathbf{U}g]$ holds:

1. The state is already labeled by formula g
2. The state is labeled by formula f and for all successors (neighbours) of the state $\mathbf{A}[f\mathbf{U}g]$ is true

In all other cases $s \vDash \mathbf{A}[f\mathbf{U}g]$ is false.

```
def au(state, formula):
    # If the state is already marked and its labels contain the formula then
    # formula holds in `state`. Otherwise we have found a circle
    # where`formula.second_subformula` is `false` or we have found a
    # successor state where formula.first_subformula` is `false`. In both
    # cases the formula does not hold in `state`
    if marked[state]:
```

```
        return True if formula in state.labels else False

    # Check if we can immediately answer the question if
    #       A [formula.first_subformula U formula.second_subformula]
    # holds in `state`
    if formula.second_subformula in state.labels:
        labels.add(formula)
        return True
    if not formula.first_subformula in state.labels:
        return False

    # For all successors excluding the current state
    for successor in state.neighbours.difference({state}):
        # For some successors state, formula does not hold
        if not au(successor, formula):
            return False
    # For all successors state `formula` holds
    return True
```

The last formula we need to check has the form $\mathbf{E}[f \mathbf{U} g]]$. Like before, we use the ideas laid out of by Clarke, Emerson, and Sistla. We start by labeling every formula where g holds and walk backwards by using the inverse of the neighbor/successor relation.

```
if formula.operator == 'EU':
    # Collect all states where the second subformula holds
    states_second_formula = [state
                             for state in kripke_structure.states
                             if formula.second_subformula in state.labels]
    # Label states and their predecessors where `formula.first_subformula`
    # holds
    for state in second_subformula:
        state.labels.add(formula)
        # For all predecessors excluding the current state
        for predecessor in state.predecessors.difference({state}):
            check_pred_eu(predecessor, formula)

def check_pred_eu(state, formula):
    if formula.first_subformula in state.labels:
        labels.add(formula)
        for predecessor in state.predecessors.difference({state}):
            check_pred(predecessor, formula)
```

3 · Bounded Model Checking

3.1 Exercise 8

Find a translation of the **U** operator to propositional logic in bounded model checking.

3.1.1 Solution

We use the procedure described in the article "Symbolic Model Checking without BDDs" [Bie+99] to translate formulas of the form $f \mathbf{U} g$ to propositional logic.

Figure 4: Path without loop

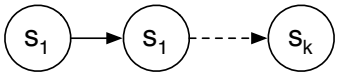
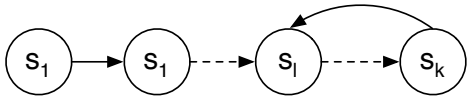


Figure 5: Path with "(k,l)-loop"



We need to distinguish between two cases. Either the path of the counterexample contains a loop (see Figure 5) or it does not (see Figure 4). For paths without a loop we define $[\![f]\!]_k^i$ as the function which translates the temporal formula f to a propositional formula. k is the bound used by the model checker and therefore describes the length of a counterexample, while i is the current position (state) in the counterexample. In addition we define ${}_l[\![f]\!]_k^i$ which does basically the same as $[\![f]\!]_k^i$, only for paths where there is a loop from state k back to state l.

Path without loop Since there is no loop in the path we just need to check if at some state s_j the formula g, and in all states before that, the formula f holds. For every possibility of j we generate a subformula. We combine these subformulas by conjunction:

$$[\![f \mathbf{U} g]\!]_k^i = \bigvee_{j=i}^{k} \left([\![g]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![f]\!]_k^n \right)$$

Path with loop We now need to consider the additional possibility that $f \mathbf{U} g$ holds on a path which starts at s_i, and

continues over the loop to end at a state before s_i.

$$ {}_l[\![f \mathbf{U} g]\!]_k^i = \bigvee_{j=i}^{k} \left({}_l[\![g]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[\![f]\!]_k^n \right) \vee \bigvee_{j=l}^{i-1} \left({}_l[\![g]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![f]\!]_k^n \wedge \bigwedge_{n=k}^{j-1} {}_l[\![f]\!]_k^n \right) $$

The translation of the rest of the temporal operators is listed in the article mentioned before [Bie+99]. For the sake of completeness we also write down the definitions here:

Translation of an LTL formula without a loop

$$ [\![p]\!]_k^i := p(s_i) \qquad [\![\neg p]\!]_k^i := \neg p(s_i) $$
$$ [\![f \wedge g]\!]_k^i := [\![f]\!]_k^i \wedge [\![g]\!]_k^i \qquad [\![f \vee g]\!]_k^i := [\![f]\!]_k^i \vee [\![g]\!]_k^i $$
$$ [\![\mathbf{G} f]\!]_k^i := false \qquad [\![\mathbf{F} f]\!]_k^i := \bigvee_{j=i}^{k} [\![f]\!]_k^j $$
$$ [\![\mathbf{X} f]\!]_k^i := \text{if } i < k \text{ then } [\![f]\!]_k^{i+1} \text{ else } false $$

Translation of an LTL formula with a loop

$$ {}_l[\![p]\!]_k^i := p(s_i) \qquad {}_l[\![\neg p]\!]_k^i := \neg p(s_i) $$
$$ {}_l[\![f \wedge g]\!]_k^i :=_l [\![f]\!]_k^i \wedge [\![g]\!]_k^i \qquad {}_l[\![f \vee g]\!]_k^i :=_l [\![f]\!]_k^i \vee [\![g]\!]_k^i $$
$$ {}_l[\![\mathbf{G} f]\!]_k^i := \bigwedge_{j=min(i,l)^k} {}_l[\![f]\!]_k^i \qquad {}_l[\![\mathbf{F} f]\!]_k^i := \bigvee_{j=min(i,l)}^{k} {}_l[\![f]\!]_k^j $$
$$ {}_l[\![\mathbf{X} f]\!]_k^i :=_l [\![f]\!]_k^{succ(i)} \qquad succ(i) := l \text{ if } i = k \text{ else } i+1 $$

We also include additional formulas we can use to translate a certain Kripke structure M, and the temporal formula f, here, to show how we are able to process loops with propositional logic.

$$ {}_lL_k = T(s_k, s_l) \qquad L_k = \bigvee_{l=0}^{k} {}_lL_k $$

$$ [\![M, f]\!]_k := [\![M]\!]_k \wedge \left(\left(\neg L_k \wedge [\![f]\!]_k^0 \right) \vee \bigvee_{l=0}^{k} \left({}_lL_k \wedge_l [\![f]\!]_k^0 \right) \right) $$

4 · Linear Temporal Logic

4.1 Exercise 9

Show that all LTL properties have counterexamples which are either finite paths or finite paths with a loop. Hint: Use the fact that LTL specifications can be translated into Büchi automata.

4.1.1 Solution

One of the standard ways to check an LTL formula is to construct a Büchi automaton for the negated LTL formula $\neg f$ and the given Kripke structure M. These two state machines accept the language $\mathscr{L}(\neg f)$ respectively $\mathscr{L}(M)$. We now construct a Kripke structure representing the intersection of the two languages. If the language accepted by this state machine is empty then $M \vDash f$ holds. On the other hand, if there exist infinite words accepted by the automaton, then these words are counterexamples for $M \vDash f$.

We now need to show that there exists either a finite path or a finite path with a loop for the language $\mathscr{L}(\neg f) \cap \mathscr{L}(M)$ if $M \nvDash f$ is true. We know that if there exists a counter-example, then there has to be a path in the Büchi automaton, where at least one acceptance state occurs infinitely often. This means that this path has to contain a loop. This implies that there has to be a finite counterexample for f, which includes a loop [Nor10].

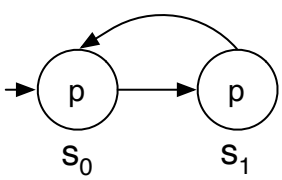
4.2 Exercise 10

Give an LTL specification where the smallest counterexample is larger than the number of states in the Kripke structure.

4.2.1 Solution

Since there is no restriction on how the Kripke structure should look, we use the one shown in Figure 6.

Figure 6: Kripke structure for exercise 10



We use the LTL specification $s_0 \vDash X(X\neg p)$. The smallest counterexample for this formula is s_0, s_1, s_0.

5 · Symbolic Model Verifier

5.1 Exercise 11

Show how you can use SMV to solve chess problems. "Given a chess board, white has a winning strategy in 3 moves." How do you describe the board? What is the specification?

5.1.1 Solution

We specify a smaller version of the problem, where we have only two chess pieces: a white rook and the black king. This has the advantage that we always know which chess piece will be moved in the next turn by either side.

To model which side has to move next, we specify the module `shared` and define the variable `next_move` inside this module. This variable alternates between the values `BLACK` and `WHITE`.

We also define the modules `king` and `rook` which define the possible movements of these two chess pieces. Each piece contains two variables `position_column` and `position_row`, which specify the position of the chess piece on the board.

In the `main` module we instantiate the `shared` module (`shared_variables`) and give a reference of this variable to the instance of the white rook (`white_rook`) and the black king (`black_king`) we defined in the lines before. We also create the variables `black_defeated` and `white_defeated` here, which tell us if one of the two sides has won. We use the variable `black_defeated` in the LTL specification

$$\neg\left((XXXX\ \neg\texttt{black\_defeated}) \wedge (XXXXX\ \texttt{black\_defeated})\right),$$

which states that there is no way that White is able to win in exactly 3 moves. If there is a counter-model to this specification, then we get a strategy where White can win in 3 moves.

The following listing shows the full NuSMV code for the simplified chess simulation.

```
-----------------------------------------------------------------------------
--  Specify a simplified simulation of a chess game
--
```

```
--  Version: 1
--  Date:    2013-09-17
--  Author:  René Schwaiger (sanssecours@f-m.fm)
-------------------------------------------------------------------------------

-- Modules ------------------------------------------------------------------

MODULE shared
    -------------------------------------------------------------------------
    -- Define data shared by both Black and White.
    -------------------------------------------------------------------------
    VAR
        -- Specifies which side has to make the next move
        next_move : {BLACK, WHITE};
    ASSIGN
        -- White always begins the game
        init(next_move) := WHITE;
        next(next_move) := case
                               next_move = BLACK: WHITE;
                               TRUE:              BLACK;
                           esac;

MODULE rook(the_color, initial_position_row, initial_position_column, shared)
    -------------------------------------------------------------------------
    -- Specify the rook chess figure.
    -------------------------------------------------------------------------
    FROZENVAR
        color : {BLACK, WHITE};
    VAR
        position_row    : 1..8;
        position_column : 1..8;
    ASSIGN
        init(color) := the_color;
        init(position_column) := initial_position_column;
        init(position_row)    := initial_position_row;
    TRANS
        case
            shared.next_move = color:
                -- A rook can either move vertically or horizontally
                (next(position_column) = position_column &
                 next(position_row) != position_row) |
```

```
                (next(position_column) != position_column &
                 next(position_row) = position_row);
            TRUE:
                next(position_row) = position_row &
                next(position_column) = position_column;
        esac;

MODULE king(the_color, initial_position_row, initial_position_column, shared)
    ------------------------------------------------------------------------
    -- Specify the king chess figure.
    ------------------------------------------------------------------------
    FROZENVAR
        color : {BLACK, WHITE};
    VAR
        position_row    : 1..8;
        position_column : 1..8;
    ASSIGN
        init(color) := the_color;
        init(position_column) := initial_position_column;
        init(position_row)    := initial_position_row;
    TRANS
        case
            shared.next_move = color:
                -- A king may move one field in any direction
                (next(position_column) = position_column - 1 &
                 next(position_row) = position_row - 1)
                |
                (next(position_column) = position_column &
                 next(position_row) = position_row + 1)
                |
                (next(position_column) = position_column + 1 &
                 next(position_row) = position_row + 1)
                |
                (next(position_column) = position_column + 1 &
                 next(position_row) = position_row)
                |
                (next(position_column) = position_column + 1 &
                 next(position_row) = position_row - 1)
                |
                (next(position_column) = position_column &
```

```
                next(position_row) = position_row - 1)
               |
               (next(position_column) = position_column - 1 &
                next(position_row) = position_row - 1)
               |
               (next(position_column) = position_column - 1 &
                next(position_row) = position_row);
           TRUE:
               next(position_column) = position_column &
               next(position_row) = position_row;
       esac;

-- Main ------------------------------------------------------------------------

MODULE main
    VAR
        shared_variables : shared;
        white_rook : rook(WHITE, 1, 1, shared_variables);
        black_king : king(BLACK, 8, 8, shared_variables);
        black_defeated : boolean;
        white_defeated : boolean;
    ASSIGN
        init(black_defeated) := FALSE;
        init(white_defeated) := FALSE;
        next(black_defeated) :=
            case
                next(white_rook.position_column) = black_king.position_column &
                next(white_rook.position_row) = black_king.position_row &
                !white_defeated: TRUE;
                TRUE:            black_defeated;
            esac;
        next(white_defeated) :=
            case
                white_rook.position_column = next(black_king.position_column) &
                white_rook.position_row = next(black_king.position_row) &
                !black_defeated: TRUE;
                TRUE:            white_defeated;
            esac;
    LTLSPEC
        -- White has a winning strategy in 3 moves
```

```
        ------------------------------------------------------------------------
        --               X  X  X  X  X     |   W....White Moves
        -- current move:  W  B  W  B  W     |   B....Black Moves
        ------------------------------------------------------------------------
        ! ((X X X X !black_defeated)  & (X X X X X black_defeated));
```

We get the following counter model for our specification which we visualized in Figure 7:

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  white_rook.color = WHITE
  black_king.color = BLACK
  shared_variables.next_move = WHITE
  white_rook.position_row = 1
  white_rook.position_column = 1
  black_king.position_row = 8
  black_king.position_column = 8
  black_defeated = FALSE
  white_defeated = FALSE
-> State: 1.2 <-
  shared_variables.next_move = BLACK
  white_rook.position_row = 2
-> State: 1.3 <-
  shared_variables.next_move = WHITE
  black_king.position_column = 7
-> State: 1.4 <-
  shared_variables.next_move = BLACK
  white_rook.position_row = 8
-> State: 1.5 <-
  shared_variables.next_move = WHITE
  black_king.position_column = 6
-- Loop starts here
-> State: 1.6 <-
  shared_variables.next_move = BLACK
  white_rook.position_column = 6
  black_defeated = TRUE
-> State: 1.7 <-
  shared_variables.next_move = WHITE
  black_king.position_column = 5
-> State: 1.8 <-
```

```
  shared_variables.next_move = BLACK
  white_rook.position_column = 4
-> State: 1.9 <-
  shared_variables.next_move = WHITE
  black_king.position_column = 6
-> State: 1.10 <-
  shared_variables.next_move = BLACK
  white_rook.position_column = 6
```

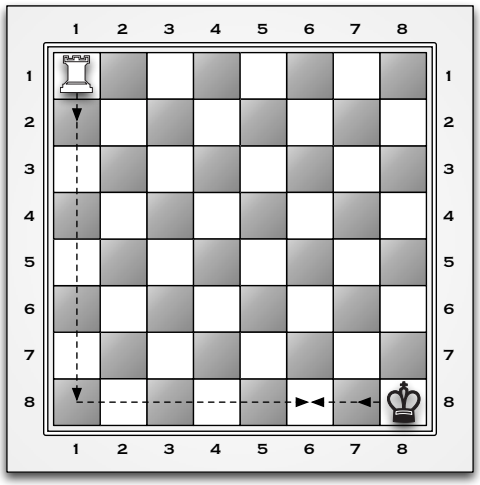


Figure 7: A (rather unlikely) way for White to win in three moves

References

[Bie+99] Armin Biere et al. ***Symbolic model checking without BDDs***. Springer, 1999.

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: ***ACM Transactions on Programming Languages and Systems (TOPLAS)*** 8.2 (1986), pp. 244–263.

[Nor10] Michael Norrish. ***COMP6463: Temporal Logic and Model Checkings***. 2010. URL: http://www.nicta.com.au/__data/assets/pdf_file/0005/19355/lecture6-ltletc.pdf.

[VHZ11] Univ. Prof. Helmut Veith, Andreas Holzer, and M.Sc. Dipl.-Math. Florian Zuleger. ***Exercises on Formal Methods in Computer Science***. Jan. 2011.