Electronic Notes in Theoretical Computer Science 47 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume47.html pp. 1 – 19

Can American Checkers be Solved by Means of Symbolic Model Checking?

Michael Baldamus
Klaus Schneider
Michael Wenz
Roberto Ziller

University of Karlsruhe
Institute for Computer Design and Fault Tolerance
Formal Methods Group
http://goethe.ira.uka.de/fmg

Abstract

Symbolic model checking has become a successful technique for verifying large finite state systems up to more than 10^{20} states. The key idea of this method is that extremely large sets can often be efficiently represented with propositional formulas. Most tools implement these formulas by means of binary decision diagrams (BDDs), which have therefore become a key data structure in modern VLSI CAD systems.

Some board games like American checkers have a state space whose size is well within the range of state space sizes that have been tackled by symbolic model checking. Moreover, the question whether there is a winning strategy in these games can be reduced to two simple μ–calculus formulas. Hence, the entire problem to solve such games can be reduced to simple μ–calculus model checking problems.

In this paper, we show how to model American checkers as a finite state system by means of BDDs. We moreover specify the existence of winning strategies for both players by simple μ–calculus formulas. Further still, we report on our experimental results with our own model checking tool, and we describe some powerful improvements that we have found in trying to solve the game.

Michael Baldamus's and Roberto Ziller's work is supported by the Deutsche Forschungsgemeinschaft (German Research Society) within the project Design and Design Methodology of Embedded Systems.

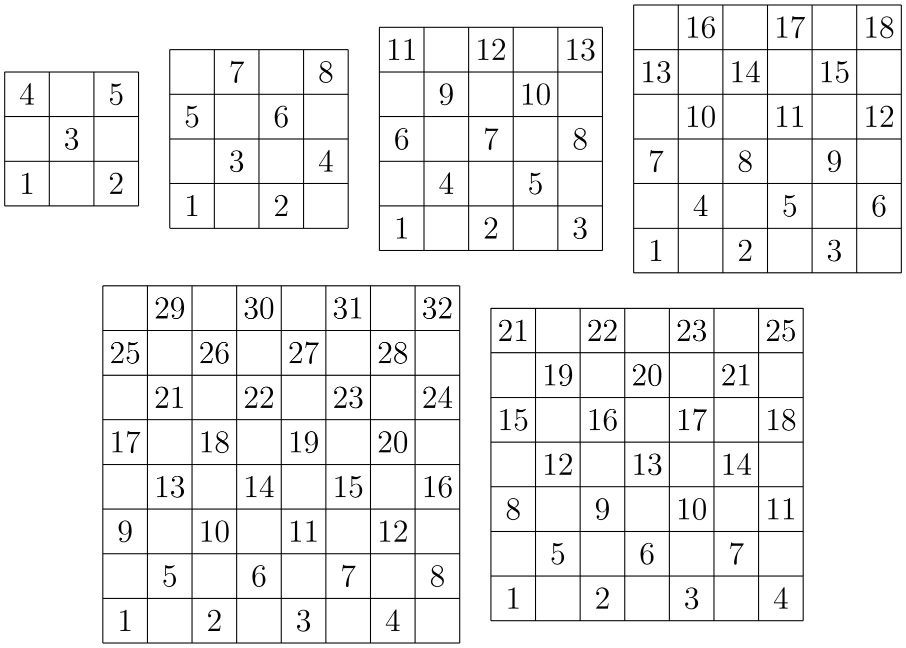

Figure 1. We consider American checkers on boards of different sizes. This figure shows numberings of the squares used in checkers for board sizes from 3×3 to 8×8.

1 Introduction

Board games such as checkers, chess or nine men's morris have a long history of many centuries. This proves the quality of these games, for most of them have survived numerous attempts to be solved, which means to either find a winning strategy for one of the players or to prove that no such strategy exists. The latter case would imply that both players are always able to conduct the game from the initial position to a draw.

Even when such a solution can be found, it is in general too complex to be used by a human player. However, from a computational point of view, this kind of solution is highly interesting, since it leads to a better understanding of practical problems.

A notable success in solving board games has been achieved by Gasser [15]. He showed that there is no winning strategy for nine men's morris. To obtain this result, Gasser implemented special search algorithms that considered a large state set. Gasser's work benefited from the fact that nine men's morris has a highly symmetric state space so that a reduction to symmetry classes reduces the problem to far fewer states.

Computers can play checkers on world champion level with a combination of brute force search and various heuristics [18]. We present an attempt at finding a solution for the game. It is remarkable that checkers does not have any symmetries that can be exploited to reduce the state space. We are therefore confronted with a much larger state space than in the case of nine men's morris. Our main idea consist of handling this large state space by symbolic traversal methods that have become popular in the domain of hardware design and finite state verification procedures. In particular, we show in this paper how the problem of finding winning strategies in checkers can be reduced to an equivalent μ–calculus verification problem (see also [20]), and we give results of our experiments. To illustrate the influence of the size of the state space on the computation time, we begin with a 3×3 board version of the game and then proceed increasing it towards the 8×8 official board size.

The paper is organized as follows: In Section 2, we present the basics of symbolic state traversal techniques and μ–calculus model checking; in Section 3, we then show how winning strategies in checkers are formulated by μ–calculus formulas, so that the problem of finding such strategies is reduced to μ–calculus model checking; in Section 4 we describe what we have done in the way of actually carrying out that model checking using automated methods. – It turned out that time and memory utilisation are both critical issues. Finally, we conclude the paper with a short summary and a discussion of possible future work.

2 Basics

One of the most challenging problems in verifying finite state systems is the *state space explosion problem*. This means that the number of states of a system may exponentially grow with the number of its components. In general, this can not be circumvented, and for this reason, there is a strong interest in algorithms that can traverse large state sets. A successful approach is known as *symbolic model checking*. This technique has been independently developed by Burch, Clarke, McMillan, and Dill [9,8,10,7], and by Berthet, Coudert, and Madre [4] (both were inspired by [2]). We give in this section a brief explanation of the basics of symbolic μ–calculus model checking. The next subsection is concerned with the use of ordered binary decision diagrams as the key data structure for these algorithms; the subsection afterward formally defines the propositional μ–calculus and a simple model checking procedure for it.

2.1 Implicit Representation of Large Sets by OBDDS

The first key idea of this approach is to store sets not by explicit enumeration. Instead, characteristic functions are used to represent sets: Recall that given a set G, the characteristic function $\chi_{G,M}$ of a subset $M \subseteq G$ is the function that maps any $e \in M$ to true, and any $e \in G \setminus M$ to false.

The second key idea is to encode the set G (which must be a finite one) by some boolean variables. Clearly, we need at least $\lfloor \log(|G|) \rfloor + 1$ such variables, but we may use more, say $\{x_1, \ldots, x_n\}$ with $n > \lfloor \log(|G|) \rfloor$. Formally, this means that we define two functions $\Phi_G : G \to \mathbb{B}^n$ and $\Phi_{\mathbb{B}} : \mathbb{B}^n \to G$, such that $\forall e \in G. e = \Phi_{\mathbb{B}}(\Phi_G(e))$ holds. With such an encoding, we can represent any characteristic function $\chi_{G,M}$ as a propositional formula $\varphi_{G,M}$ over the variables $\{x_1, \ldots, x_n\}$ such that $\varphi_{G,M}$ is satisfied by a minterm whenever this minterm encodes a member of the set M. As all equivalent formulas obviously encode the same set, we are particularly interested in propositional normal forms.

While there are many normal forms for propositional formulas that could all be used in this setting, OBDDs as developed by Bryant [5] are best suited for this purpose. OBDDs are based on the so–called Shannon decomposition of a propositional formula φ, which says that a formula φ is equivalent to $x \wedge \varphi_{x:=1} \vee \neg x \wedge \varphi_{x:=0}$, denoting by $\varphi_{x:=\tau}$ the formula obtained by replacing x with τ in φ. This decomposition may be viewed as a case distinction where in the first case, it is assumed that x is true, so that $\varphi = \varphi_{x:=1}$ holds, and in the second case, it is assumed that x is false, so that $\varphi = \varphi_{x:=0}$ holds. The formulas $\varphi_{x:=1}$ and $\varphi_{x:=0}$ are sometimes called the cofactors of φ.

An OBDD of a formula is obtained by a complete Shannon decomposition until only constants are obtained as cofactors. Bryant's observation was that OBDDs provide a normal form for propositional formulas whenever the Shannon decomposition is done in the same order for all cofactors. — Hence, they are called *ordered* BDDs, or OBDDs for short. In the following, we only

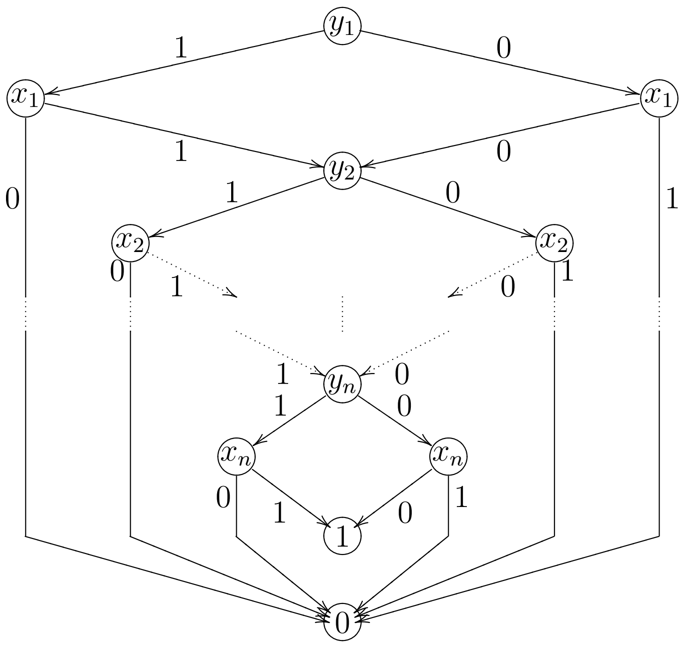

Figure 2. BDD for $\bigwedge_{i=1}^{n} x_i \equiv y_i$.

consider OBDDs, so we simply speak of BDDs.

BDDs can be efficiently implemented and allow us therefore to manipulate large propositional formulas. Hash–tables are used to keep track of previously computed BDDs, so that recomputations are avoided, and common sub–BDDs are shared. For this reason, BDDs may be pictorially represented as acyclic graphs as given in Figure 2 with a single root vertex. Vertices of these graphs are also often called nodes. We denote the number of nodes of a BDD B as $|B|$.

Tautologies are all reduced to the leaf node 1, and unsatisfiable formulas to the leaf node 0. For any other formula, we obtain a model of the formula by following any path from the root to a leaf 1.

Using special algorithms given in [5], we can moreover efficiently perform boolean operations such as conjunction or disjunction on BDDs B_1 and B_2: the result will have at most $|B_1||B_2|$ nodes. On the other hand, BDDs may still suffer from an exponential blow–up (as any normal form will necessarily do [12]). The size of a BDD crucially depends on the ordering in which the Shannon decomposition is done (also simply called variable orderings). Different variable orderings lead to different BDDs that may considerably differ in their size. For example, any BDD for the formula $\bigwedge_{i=1}^{n} x_i \equiv y_i$ with an ordering where all x_i's appear before any y_i will have an exponential size in n while the one given in Figure 2 has size $3n + 2$.

Today, BDDs are used in many applications, in particular to solve graph based problems. Many variants using different decomposition schemes have been developed [3,13]. A couple of publicly available BDD packages can be downloaded from the Internet [19].

2.2 Symbolic Model Checking of the Propositional μ–Calculus

Most verification problems, including the verification of temporal logic specifications, can be reduced to equivalent fixed point problems. This view has been advocated by Emerson and Lei [14] early on in that they have shown how various problems can be naturally expressed using the propositional μ–calculus [16].

Definition 2.1 (Syntax of μ–Calculus) The following rules define the set of μ–calculus formulas $\mathcal{L}_\mu$ over a given finite set of variables $\mathcal{V}$:

Variables: each variable is a μ–calculus formula: $\mathcal{V} \subseteq \mathcal{L}_\mu$
Closure under Boolean Operators: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi \in \mathcal{L}_\mu$ if $\varphi, \psi \in \mathcal{L}_\mu$
Closure under Next–State Operators: $\boldsymbol{EX}\varphi$, $\boldsymbol{AX}\varphi \in \mathcal{L}_\mu$ if $\varphi \in \mathcal{L}_\mu$
Closure under Fixed Point Operator: $\mu\, x.\varphi \in \mathcal{L}_\mu$ if $x \in \mathcal{V}$ and $\varphi \in \mathcal{L}_\mu$

We must additionally impose the restriction on $\mathcal{L}_\mu$ formulas that in case of fixed point formulas $\mu x.\varphi$, all occurrences of x in φ must be positive, that is, occur beneath an even number of negation symbols. This requirement assures that fixed points exist, and in particular that least fixed points exist. The semantics is given with respect to Kripke structures, which are sometimes also simply called labelled transition systems.

Definition 2.2 (Kripke Structures) A Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ for a set of variables $\mathcal{V}$ is given by a finite set of states $\mathcal{S}$, a set of initial states $\mathcal{I} \subseteq \mathcal{S}$, a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and a labeling function $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{V}}$ that maps each state to a set of variables.

To define the semantics of μ–calculus formulas, we need to define the existential and the universal predecessor sets of a set of states $Q \subseteq \mathcal{S}$. These are formally defined as:

- $\boldsymbol{pre}_\exists^{\mathcal{R}}(Q) := \{s_1 \in \mathcal{S} \mid \exists s_2.(s_1, s_2) \in \mathcal{R} \wedge s_2 \in Q\}$
- $\boldsymbol{pre}_\forall^{\mathcal{R}}(Q) := \{s_1 \in \mathcal{S} \mid \forall s_2.(s_1, s_2) \in \mathcal{R} \implies s_2 \in Q\}$

$\boldsymbol{pre}_\exists^{\mathcal{R}}(Q)$ is the set of all states that have at least one successor state in Q; and $\boldsymbol{pre}_\forall^{\mathcal{R}}(Q)$ is the set of all states where all successor states belong to Q. To define the semantics, we need one more piece of notation: Given a Kripke structure $\mathcal{K}$, a variable x and a set of states $Q \subseteq \mathcal{S}$, we denote the structure where exactly the states in Q are labelled with x as $\mathcal{K}_x^Q$. All other labels are retained in the modified structure $\mathcal{K}_x^Q$.

Definition 2.3 (Satisfying States of a Formula) Given a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, we define the set of satisfying states of a state formula Φ as follows:

- $[\![x]\!]_{\mathcal{K}} := \{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$ for all variables $x \in \mathcal{V}$
- $[\![\neg\varphi]\!]_{\mathcal{K}} := \mathcal{S} \setminus [\![\varphi]\!]_{\mathcal{K}}$

- $[\![\varphi \wedge \psi]\!]_{\mathcal{K}} := [\![\varphi]\!]_{\mathcal{K}} \cap [\![\psi]\!]_{\mathcal{K}}$
- $[\![\varphi \vee \psi]\!]_{\mathcal{K}} := [\![\varphi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}}$
- $[\![\boldsymbol{EX}\,\varphi]\!]_{\mathcal{K}} := \boldsymbol{pre}_{\exists}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\boldsymbol{AX}\,\varphi]\!]_{\mathcal{K}} := \boldsymbol{pre}_{\forall}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
- $[\![\mu\, x.\varphi]\!]_{\mathcal{K}} := \bigcap\{Q \subseteq \mathcal{S} \mid [\![\varphi]\!]_{\mathcal{K}_x^Q} \subseteq Q\}$

One can easily verify that $[\![\mu\, x.\varphi]\!]_{\mathcal{K}}$ is the least set of states $Q \subseteq \mathcal{S}$ such that $Q = [\![\varphi]\!]_{\mathcal{K}_x^Q}$ holds. Moreover, with the requirement that all occurrences of x in φ are positive, it follows that such a fixed point always exists, and that there is always a (uniquely determined) least one.

The set of states that satisfy a μ–calculus formula φ can be computed by symbolic model checking using Tarski's famous fixed point theorem [21]. A simple algorithm is given in Figure 3. It is easily seen that the algorithm runs in exponential time in case there are nested fixed point operators. This can be significantly improved [11]. However, the formulas we need to check in order to find winning strategies do not have nested fixed point operators, and therefore the algorithm of Figure 3 is well suited for our purposes.

```
function $[\![\Phi]\!]_{\mathcal{K}}$
  case $\Phi$ of
    $\boldsymbol{is\_var}(\Phi)$ : return $\{s \mid \Phi \in \mathcal{L}(s)\}$;
    $\neg\varphi$         : return $\mathcal{S} \setminus [\![\varphi]\!]_{\mathcal{K}}$;
    $\varphi \wedge \psi$       : return $[\![\varphi]\!]_{\mathcal{K}} \cap [\![\psi]\!]_{\mathcal{K}}$;
    $\varphi \vee \psi$       : return $[\![\varphi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}}$;
    $\boldsymbol{EX}\,\varphi$       : return $\boldsymbol{pre}_{\exists}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$;
    $\boldsymbol{AX}\,\varphi$       : return $\boldsymbol{pre}_{\forall}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$;
    $\mu\, x.\varphi$       : $Q_1 := \{\}$;
                  repeat
                     $Q_0 := Q_1$;
                     $\mathcal{L} := \mathcal{L}_x^{Q_1}$;
                     $Q_1 := [\![\varphi]\!]_{\mathcal{K}}$;
                  until $Q_0 = Q_1$;
                  return $Q_0$;
end;
```

Figure 3. Algorithm for Checking μ calculus formulas

It is easily seen that the set operations can be replaced with corresponding boolean operations when a symbolic encoding with BDDs is used. The computation of $[\![\boldsymbol{AX}\,\varphi]\!]_{\mathcal{K}}$ can be reduced to the computation of $[\![\boldsymbol{EX}\,\varphi]\!]_{\mathcal{K}}$, since we have $\boldsymbol{AX}\,\varphi \equiv \neg\, \boldsymbol{EX}\,\neg\varphi$. Hence, the only problem that remains is to compute the function $\boldsymbol{pre}_{\exists}^{\mathcal{R}}(Q)$.

To see how this is implemented with BDDs, assume that the states of a particular Kripke structure $\mathcal{K}$ are encoded with some boolean state variables, say $\{x_1, \dots, x_n\}$. The transition relation can then be represented by a propositional formula $\mathcal{R}(x_1, \dots, x_n, x'_1, \dots, x'_n)$ over variables $\{x_1, \dots, x_n, x'_1, \dots, x'_n\}$, where $x_1, \dots, x_n$ encode a current state and $x'_1, \dots, x'_n$ a successor state. There is a transition from a state encoded by $x'_1, \dots, x'_n$ to another one encoded by $x'_1, \dots, x'_n$ iff the formula $\mathcal{R}(x_1, \dots, x_n, x'_1, \dots, x'_n)$ is evaluated to true under this assignment of variables. In a similar way, we can represent sets of states by propositional formulas over the variables $\{x_1, \dots, x_n\}$. Now, assume that $Q(x_1, \dots, x_n)$ represents a set of states. Then, we can compute the set of existential predecessors $\boldsymbol{pre}_{\exists}^{\mathcal{R}}(Q)$ with BDDs as follows:

$$\boldsymbol{pre}_{\exists}^{\mathcal{R}}(Q) := \exists\, x'_1 \dots x'_n . Q(x_1, \dots, x_n) \wedge \mathcal{R}(x_1, \dots, x_n, x'_1, \dots, x'_n)$$

Existential quantification over a boolean variable is thereby defined as $\exists x.\varphi := \varphi_{x:=0} \vee \varphi_{x:=1}$. It is important to implement the above 'relational product' efficiently, since it is frequently used in the model checking algorithm given in Figure 3. There are a lot of state–of–the–art techniques to do that. We will discuss this issue in more detail in Section 4.

3 Reducing the Checkers Problem to μ–Calculus Model Checking

3.1 Rules

Checkers is played in many variations around the world. The rules presented below are extracted from the web site of the American Checkers Federation [1].

I. Checkers is played on the dark fields of an 8×8–grid of light and dark square fields. The board is placed between the players in such a way that each of them has a light square on his or her near right corner. Each player places his or her pieces on the dark squares of the rows nearest to him or her. The player with the darker pieces makes the first move of the game, and the players take turns thereafter, making one move at a time.

II. The objective of the game is to prevent the opponent from moving, either by capturing all of his or her pieces or by blocking them. If neither player can accomplish this, the game is a draw.

III. Single pieces, known as men, move forward only, one square at a time in diagonal direction. They can capture any opposing checker on a diagonally adjacent square by jumping over it to a free square immediately beyond. Jumping continues as part of the same move as long as there are adjacent opposing pieces with a free square beyond them. Men may not jump over pieces of their own colour.

IV. A man that reaches the far side of the board becomes a king. The opponent must crown the new king before making his or her next move. This implies that a new king cannot continue jumping in the same turn it became crowned.

V. Kings move and jump forward or backward. They may not jump over pieces of the same colour and may not jump over the same opposing man or king more than once.

VI. When a player is able to make a capture, he or she *must* do so. If there is more than one way to capture, the player can choose any possibility. However, all the possible captures in the chosen sequence must be made.

3.2 Modelling the Game

According to the methodology presented in Section 2, constructing a model for the game amounts to defining the associated Kripke structure and set of variables. An automatic test tool can then verify whether a given assertion in the form of a μ–calculus formula — for example the statement that black (or white) can always win — is true for the given model.

In order to define the state space of the Kripke structure, we use a set of variables that reflect all situations that can possibly be achieved during a game. We choose the variables considering the different states in which a square of the board can be. In addition to being empty, a square can have a man or a king, which can be black or white. Another important consideration that comes from rule VI is whether a man or king must go on capturing during its turn. While in the real game jumping over many pieces is considered a single move, we model each jump as a single transition. The equivalence to the real game is guaranteed through the fact that we do not toggle turns while capturing must go on.

The possible states associated with each square are therefore: 'empty', 'black man', 'white man', 'man must capture', 'black king', 'white king', and 'king must capture'. These seven possible states for the square i, j can be encoded with three binary variables $x_{i,j}$, $y_{i,j}$, and $z_{i,j}$ (with one unused combination). An additional variable m tells which player has to move. This information is also used in conjunction with the states where a man or a king must capture, in order to determine its colour. Thus, the number of binary variables for a board of size $n \times n$ with d dark squares is

$$3 \times d + 1, \text{ where } d \text{ is given by } d = \begin{cases} \frac{1}{2}n^2 & n \text{ even} \\ \frac{1}{2}(n^2+1) & n \text{ odd.} \end{cases}$$

We still need a copy of each variable to represent the current state and the next states in the transition relation of the Kripke structure. Hence, the total number of binary variables is twice the number of variables used to represent the states of the game. We moreover immediately see that there are at most $2^{\frac{3}{2}(n^2+1)+1}$ possible game situations.

3.2.1 Defining the State Space and the Initial State

Each state in the Kripke structure corresponds to one of the possible value assignments to the set of variables. This allows us to define the state set of the Kripke structure as the set of all possible value assignments that can be made to the chosen set of variables; we do not list all of those combinations explicitly, since this is clearly impracticable. The variable assignments corresponding to the initial state reflect the starting position of the game with m telling that black has to play.

3.2.2 Defining the Transition Relation

The transition relation $\phi_{checkers}$ for the Kripke structure can now be given as a boolean formula that takes into account all the possible moves that can happen from each square. For example, the fact that a black man can move from the leftmost lower corner to the diagonally adjacent square is stated like:

- the leftmost lower square has a black man
- and the diagonally adjacent square is empty
- and in the next state the leftmost lower square will be empty
- and in the next state the diagonally adjacent corner will have a black man
- and it is black's turn
- and in the next state it will not be black's turn
- and black can not capture
- and the next state for all other squares remains equal to the current state
- and it is not already a winning state for either black or white

(terms like "its not already a winning state" are also expanded into Boolean assertions about the defined variables). This completes the definition of the Kripke structure. The next step is to write down the formulas that express the behaviour we want to verify in the given model. Two different statements must be made to find out whether there is a winning strategy for black or white. In the formulas presented below, $\phi_{black\_has\_won}$ is expanded into the Boolean assertion "not black's turn and white can not move and white can not capture"; $\phi_{white\_has\_won}$ is the corresponding term for white. The formula that specifies a winning strategy for black, $\phi_{black\_can\_win}$, is

$$\boxed{\mu\, x.\phi_{black\_has\_won} \vee (m \wedge \boldsymbol{EX}\, x) \vee (\neg m \wedge \boldsymbol{AX}\, x),} \quad \boxed{\phi_{black\_can\_win}}$$

meaning that:

- the state is an end–state in which black has won ($\phi_{black\_has\_won}$) or
- it is black's turn ($m = 1$) and there is a next state in which the formula is recursively true for at least one next move ($\boldsymbol{EX}\, x$) or
- it is white's turn ($m = 0$) and the formula is recursively true for all possible next moves ($\boldsymbol{AX}\, x$)

The formula that specifies a winning strategy for white, $\phi_{\mathit{white\_can\_win}}$, is defined analogously:

$$\boxed{\mu\, x.\phi_{\mathit{white\_has\_won}} \lor (\neg m \land \boldsymbol{EX}\, x) \lor (m \land \boldsymbol{AX}\, x) \quad \boxed{\phi_{\mathit{white\_can\_win}}}}$$

The formulas $\phi_{\mathit{black\_has\_won}}$ and $\phi_{\mathit{white\_has\_won}}$ can also be easily defined by μ–calculus formulas: $\phi_{\mathit{black\_has\_won}} := \neg m \land \boldsymbol{AX}\, 0$ and $\phi_{\mathit{white\_has\_won}} := m \land \boldsymbol{AX}\, 0$. To see that these definitions are correct, note that $[\![\boldsymbol{AX}\, 0]\!]_{\mathcal{K}}$ is the set of states of $\mathcal{K}$ that have no outgoing transition. Hence, $\phi_{\mathit{black\_has\_won}}$ means that it is white's turn ($m = 0$), but there is no possibility to proceed with the game, and analogously for $\phi_{\mathit{white\_has\_won}}$. Using these definitions of $\phi_{\mathit{black\_has\_won}}$ and $\phi_{\mathit{white\_has\_won}}$, we can furthermore simplify our μ–calculus formulas as follows:

$$\begin{aligned}
\phi_{\mathit{white\_can\_win}} &\equiv \mu x.\phi_{\mathit{white\_has\_won}} \lor (\neg m \land \boldsymbol{EX}\, x) \lor (m \land \boldsymbol{AX}\, x) \\
&\equiv \mu x.(m \land \boldsymbol{AX}\, 0) \lor (\neg m \land \boldsymbol{EX}\, x) \lor (m \land \boldsymbol{AX}\, x) \\
&\equiv \mu x.(m \land (\boldsymbol{AX}\, 0 \lor \boldsymbol{AX}\, x)) \lor (\neg m \land \boldsymbol{EX}\, x) \\
&\equiv \mu x.(m \land \boldsymbol{AX}\, x) \lor (\neg m \land \boldsymbol{EX}\, x),
\end{aligned}$$

using the law $\boldsymbol{AX}\, 0 \lor \boldsymbol{AX}\, \phi \equiv \boldsymbol{AX}\, \phi$. With this formalisation of winning strategies as μ–calculus formulas, the problem is now reduced to μ–calculus model checking. This means that all we have to do is to use the algorithm given in Figure 3 to compute the set of states $[\![\phi_{\mathit{black\_can\_win}}]\!]_{\mathcal{K}_{\mathit{checkers}}}$, where $\mathcal{K}_{\mathit{checkers}}$ is the Kripke structure for the checkers game, and $\phi_{\mathit{black\_can\_win}}$ is the above–defined formula. Note that $[\![\phi_{\mathit{black\_can\_win}}]\!]_{\mathcal{K}_{\mathit{checkers}}}$ is the set of states where black has a winning strategy. In particular, this is the case for the states where black has won, hence, we clearly have $[\![\phi_{\mathit{black\_has\_won}}]\!]_{\mathcal{K}_{\mathit{checkers}}} \subseteq [\![\phi_{\mathit{black\_can\_win}}]\!]_{\mathcal{K}_{\mathit{checkers}}}$, so this set is certainly not empty. The important question is, however, whether the initial position of the game belongs to either $[\![\phi_{\mathit{black\_can\_win}}]\!]_{\mathcal{K}_{\mathit{checkers}}}$ or $[\![\phi_{\mathit{white\_can\_win}}]\!]_{\mathcal{K}_{\mathit{checkers}}}$, which would mean that either black or white has a winning strategy from the beginning.

4 Carrying Out the Model Checking Using Automated Methods

In Section 2 we have recalled the basics of the symbolic model checking of μ–formulas; in Section 3 we have described a reduction of the checkers problem to the satisfaction of two specific μ–formulas. This section combines these strands: We describe how we have used automated symbolic methods in trying to solve the verification task set out in Section 3.

4.1 Building the BDD Representation of $\phi_{checkers}$

To begin with, we consider the BDD representation of checkers, that is, the BDD normal form of the transition relation of checkers given in the preceding section. This sub–task has to be completed for all attempts to solve checkers by symbolic traversal methods. The aim is to find a representation with as few BDD nodes as possible.

Several values can be found in Table 1, as we have not only considered the 8 × 8–board but also smaller boards starting from a 3 × 3–board. It is thereby possible to find out just how far one can get in solving checkers through symbolic model checking.

All values are stated for both a fixed and a heuristically optimised variable ordering. The fixed ordering is respectively called the *horizontal* ordering and was chosen for simplicity. It is given by

$$\begin{aligned} m \quad \prec \quad & x_{i_1} \prec x'_{i_1} \prec y_{i_1} \prec y'_{i_1} \prec z_{i_1} \prec z'_{i_1} \quad \prec \quad \ldots \\ & \ldots \quad \prec \quad x_{i_n} \prec x'_{i_n} \prec y_{i_n} \prec y'_{i_n} \prec z_{i_n} \prec z'_{i_n}, \end{aligned}$$

where n is the number of black squares, $i_1, \ldots, i_n$ is the numbering of these squares as it is depicted in Figure 1 and "$'$" designates a next state variable. The optimised ordering was in all cases computed by the built–in *sifting algorithm* of the SMV model checker.

	variable ordering	
	horizontal	sifted
3×3	482	482
4×4	$3,165$	$2,966$
5×5	$\approx 36,000$	$\approx 34,000$
6×6	$\approx 138,000$	$\approx 119,000$
7×7	$\approx 1,295,000$	$\approx 471,000$
8×8	$\approx 4,601,000$	$\approx 1,594,000$

Table 1
BDD sizes in representing checkers.

As for interpreting these figures, the tool that we have used for trying to solve checkers needs 16 bytes to store each BDD node (cf. Figure 4 on the next page). It is thus possible to represent checkers in 74 MB of main memory, given that one uses the horizontal variable ordering. Only 26 MB are required if one uses the ordering computed by the SMV tool.

It became clear that we could not solve checkers symbolically without optimising the basic algorithm. This situation, in turn, required tight control, so we developed SymQuest, a symbolic model checker for alternation–free μ–formulas. SymQuest's input language is a subset of the input language of the popular symbolic model checker SMV, up to that SymQuest accepts μ–formulas. The input language overlap has of course helped validating SymQuest.

The core of a symbolic model checker is always made up of a BDD package. We did not develop our own package but relied on the Cal BDD package [17].

Figure 4. SymQuest — a verification tool.

4.2 Verifying $\phi_{black\_can\_win}$ and $\phi_{white\_can\_win}$

4.2.1 Failure of Straightforward Symbolic Model Checking

For our experiments, we mustered a PC with 1 GB of main memory. Two facts gave hope that we might be able to solve checkers: first, the possibility of using just 26 MB to represent the game; second, 500,995,484,682,338,672,639 is the number of positions according to [22], a figure well within the range of state space sizes tackled with symbolic model checking to date. Our next step was therefore to compute $[\![\phi_{black\_can\_win}]\!]_{\mathcal{K}_{checkers}}$ or $[\![\phi_{white\_can\_win}]\!]_{\mathcal{K}_{checkers}}$ right away, without any optimisations. The result was disappointing, as the verification process got stuck due to memory overflow. Using swap space is not an option; the reason is that BDD packages slow down drastically once swapping occurs, confirming a general experience with the runtime behaviour of BDD algorithms.

4.2.2 Investigating Various Optimisation Heuristics

Because of the above–described failure, we concluded that the decisive issue was to reduce the peak number of BDD nodes allocated during verification. This proposition, in turn, led us to investigating various techniques that can influence BDDs sizes in symbolic model checking.

Restricting Model Checking to Reachable States

One technique consists of restricting model checking to reachable states. It depends on the particularities of the respective verification task whether the BDDs involved become smaller in this way, but often they do become smaller.

As far as our purposes are concerned, this approach requires a BDD that represents the set of reachable states. This BDD can also be subject to the size problem. It turns out that checkers is a case where this calamity occurs. Specifically, we have not been able to build the BDD for the set of reachable states beyond the 5×5–case (cf. Table 2). We concluded that restricting model checking to reachable states is, therefore, not a road to success.

3×3	70
4×4	2081
5×5	$\approx 1,400,000$

Table 2
BDD sizes in representing the reachable states of checkers using SMV with sifting turned on.

Ruling out States with Too Many Pieces

A technique that is similar to restricting model checking to reachable states consists of ruling out at least some unreachable states, again hoping that the ensuing BDDs become smaller. We have found such a technique; it leads to a significant reduction of the peak number of BDD nodes at least in the 3×3– and 4×4–cases. What is more, the peak number in these cases is actually smaller than what is achieved by the restriction to reachable states. We expect that this overall situation also prevails in all other cases.

The idea is simply to rule out (unreachable) states that contain more black or white pieces than are theoretically possible; on an 8×8 board, for example, at most 12 pieces of each color are possible. Technically, let ϕ_{limit} be a formula that characterises all (reachable or un–reachable) states with at most p black pieces and at most p white pieces, where p is the number of pieces of each color at the beginning. The model checking algorithm is so modified as to substitute the formula $\phi_{checkers} \wedge \phi_{limit}$ for the formula $\phi_{checkers}$ at all places. Concrete results are such that the peak number of BDD nodes in verifying 4×4–checkers is reduced from $90,125$ to $19,725$ relative to the horizontal variable ordering. Another benefit is a reduction of the number of fixed point iterations from 32 to 16.

Finding a Good Variable Ordering

Up to this point, we had not devoted any serious attention to the issue of finding a variable ordering that was as good as possible. There are lots of heuristics for this task, but we adopted a brute force method. What we did was to consider all permutations of the black squares of the 4×4–board; we verified $\phi_{black\_can\_win}$ with each one of the corresponding variable orderings, analysing what orderings led to the lowest peak in BDD node usage. This program was manageable because of the short runtime of the 4×4–case on a 500 Mhz PC combined with the fact that there are only $40,320$ permutations of those 12 squares. — We only considered placing m at the beginning or at the end of the ordering. Also, our orderings were always such that x_i, x'_i, y_i y'_i, z_i and z'_i occurred in this same order and without any interspersed variables for each square i. To our experience, tearing apart such blocks of closely connected variables is almost never advantageous.

The result was that all of the 100 best orderings were variations of a unique diagonal pattern, and that this pattern itself was only marginally worse than the best ordering. The pattern is shown in Figure 5. It reduces the above–stated peak of $19,725$ to $16,415$ when m is placed at the beginning. We expect that analogous orderings are also good for verifying $\phi_{black\_can\_win}$ and $\phi_{white\_can\_win}$ for the other board sizes.

	2		1
5		3	
	6		4
8		7	

Figure 5. Diagonal square ordering.

On a more intuitive remark, it is not unusual in symbolic model checking that problems with some kind of regular structure are best encoded with some kind of regular variable ordering. This situation seems to prevail in the case of American checkers as well. Specifically, the game has a regular structure in the sense that the board is regular and in the sense that only a small number of adjacent squares are directly involved in each individual move, where the rules governing this move are always the same up to the possible promotion of a man to a king. Then, the diagonal variable ordering is obviously regular too.

Partitioning the BDD Used to Represent $\phi_{checkers}$

Still another possibility of reducing the peak number of BDD nodes consists of partitioning the BDD used to represent $\phi_{checkers}$ (cf. [6]).

The background consists in part of the fact that the most costly BDD operation that we need is the relational product of $\phi_{checkers}$ and ψ (cf. Section 2).; its (exponential) worst case complexity is $|BDD(\chi))|^{|X|}$, where χ is the formula beneath the quantifier, $BDD(\chi)$ is its BDD representation and X is the set of quantified variables.

The other background aspect of partitioning is that $\phi_{checkers}$ has the form

$$\phi_1 \vee \cdots \vee \phi_k$$

for some $k \geq 1$, where every ϕ_i describes what moves can occur on two specific, diagonally adjacent squares s_1 and s_2, or what jumps can occur on three such squares s_1, s_2 and s_3, where a piece on s_2 is taken. Every ϕ_i is given as

$$\phi_i := \phi_{i,change} \wedge \phi_{i,stable}.$$

The first formula, $\phi_{i,change}$, is a formula over m, m' and the variables associated with s_1 and s_2 or with s_1, s_2 and s_3; this formula describes the actual moves or jumps. The second formula, $\phi_{i,change}$, is a formula over all other variables; this formula is a conjunction of equivalences of the form $x \equiv x'$, which determine that those moves or jumps leave the rest of the board unaffected.

The idea is to rewrite the relational product of $\phi_{checkers}$ and ψ so that existential quantification occurs not once over all but several times over some next state variables. To this end, we exploit that existential quantification distributes over disjunction, meaning that

$$\exists \boldsymbol{s}'.\phi_{checkers} \wedge \psi[\boldsymbol{s}'/\boldsymbol{s}] \equiv (\exists \boldsymbol{s}'.\phi_1 \wedge \psi[\boldsymbol{s}'/\boldsymbol{s}]) \wedge \cdots \wedge (\exists \boldsymbol{s}'.\phi_k \wedge \psi[\boldsymbol{s}'/\boldsymbol{s}]).$$

We also exploit that substituting x' for x in ψ and quantifying over it is unnecessary if $\phi_{i,stable}$ determines $x \equiv x'$; this property entails

$$\exists \boldsymbol{s}'.\phi_i \wedge \psi[\boldsymbol{s}'/\boldsymbol{s}] \equiv \exists \boldsymbol{s}'_{i,change}.\phi_{i,change} \wedge \psi[\boldsymbol{s}'_{i,change}/\boldsymbol{s}_{i,change}],$$

where $\boldsymbol{s}'_{i,change}$ is the vector of free variables of $\phi_{i,change}$, $1 \leq i \leq k$.

In sum, each quantification involves 7 or 10 variables regardless of the board size, and the formula beneath the quantifier does not involve $\phi_{checkers}$ but $\phi_{i,change}$, $1 \leq i \leq k$. What is more, such a $\phi_{i,change}$ is only moderately complex and also independent of the board size up to variable renaming; for this reason, its BDD representation is much smaller than that of $\phi_{checkers}$. The effect is another reduction of the space and time requirements of verifying $\phi_{black\_can\_win}$ and $\phi_{white\_can\_win}$. A particular advantage is that building the BDD representation of $\phi_{checkers}$ becomes unnecessary; it suffices to build the BDD representation of each individual $\phi_{i,change}$. This situation is the reason why one speaks of partitioning, or of disjunctive partitioning, as far as our context is concerned.

A tradeoff involved in partitioning consists of the time and space necessary to form the BDD representation of

$$\bigvee_{i=1}^{k} \exists \boldsymbol{s}'_{i,change}.\phi_{i,change} \wedge \psi[\boldsymbol{s}'_{i,change}/\boldsymbol{s}_{i,change}]$$

in terms of the representation of each individual disjunct $\exists \boldsymbol{s}'_{i,change}.\phi_{i,change} \wedge \psi[\boldsymbol{s}'_{i,change}/\boldsymbol{s}_{i,change}]$. Working with fewer partitions may in general be more time but less space efficient. Indeed, our most space efficient run for the 4×4–board involves full partitioning, has a peak of $11,945$ nodes and requires 37.09 seconds of user time on a 500 MHz Pentium III PC under Linux. The user time can be reduced to $15,77$ seconds by keeping all options but using 10 partitions. In this case the peak is $16,180$.

Putting Everything Together

Finally, putting everything together means to use our optimisations all at once, that is, replacing $\phi_{checkers}$ by $\phi_{checkers} \wedge \phi_{limit}$, using the diagonal variable ordering and partitioning $\phi_{checkers} \wedge \phi_{limit}$, which is possible since $\phi_{checkers} \wedge \phi_{limit} \equiv (\phi_1 \wedge \phi_{limit}) \vee \cdots \vee (\phi_k \wedge \phi_{limit})$. The above–stated peak of $11,945$ nodes is actually the outcome of that. Specifically, our experimental results indicate that $11,945$ nodes is the best 4×4–peak that those optimisations can achieve.

5 Conclusions

We have explored the question whether American checkers can be solved by means of symbolic model checking of μ–formula. To this end, we have first described how the game can be represented with a boolean formula; second, we have described what those μ–formulas look like; third, we have reported our failure in terms of straightforward model checking; fourth, we have examined the effect of various optimisations on the verification process.

The next question is how far our optimisations can get us with respect to board sizes bigger than 4×4. Here, we mention that a 5×5–run without optimisations got stuck for days in the fourth iteration before we terminated it; with the maximum number of partitions and the other optimisations, the average over several weeks is between one and two days per iteration. Moreover, memory utilisation stabilises at a staggering 500 MB. At far lower levels, this kind of stabilisation can also be observed in the 3×3– and 4×4–cases. For the record, our results in terms of solving the game are stated in Figure 6. The total user time needed to solve the 5×5–case was about two months on platform mentioned in Section 4.

3×3	white can win
4×4	a draw
5×5	black can win
$6 \times 6 - 8 \times 8$	?

Figure 6. Final results. Note that the 3×3–case can very easily be solved without a computer.

We conclude two things: First, our optimisations scale up as the 5×5–case becomes solvable if one is patient enough; second, the jump from 4×4 to 5×5 in terms of memory utilisation is so drastic that it is not obvious how one could get beyond 5×5, even if time utilisation were not an issue. The checkers problem thus seems to be a very hard and peculiar one if one tries to solve it with symbolic model checking.

References

[1] American Checkers Federation, *Official Website* (May 2000), `http://www.acfcheckers.com`.

[2] Beatty, D., R. Bryant and C.–J. Seger, *Synchronous Circuit Verification by Symbolic Simulation: an Illustration*, in: W. Dally, editor, *Advanced Research in VLSI* (1990), pp. 98–112, conference proceedings.

[3] Becker, B. and K. Drechsler, *How Many Decomposition Types Do We Need?*, in: *Design and Test* (1995), pp. 438–443, proceedings EDTC '95.

[4] Berthet, C., O. Coudert and J. Madre, *New Ideas on Symbolic Manipulation of Finite State Machines*, in: *Computer Aided Design*, 1990, proceedings ICCAD '90.

[5] Bryant, R., *Graph–Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers **C–35** (1986), pp. 677–691.

[6] Burch, J., E. Clarke and D. Long, *Representing Circuits More Efficiently in Symbolic Model Checking*, in: *Design Automation*, 1991, pp. 403–407, proceedings DAC '91.

[7] Burch, J., E. Clarke, D. Long, K. McMillan and D. Dill, *Symbolic Model Checking for Sequential Circuit Verification*, IEEE Transactions on Computer–Aided Design of Integrated Cicuits and Systems **13** (1994), pp. 401–424.

[8] Burch, J., E. Clarke, K. McMillan and D. Dill, *Sequential Cicuit Verification Using Symbolic Model Checking*, in: *Design Automation* (1990), pp. 46–51, proceedings DAC '90.

[9] Burch, J., E. Clarke, K. McMillan, D. Dill and L. Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, in: *Logic in Computer Science* (1990), pp. 1–33, proceedings LICS '90 symposium.

[10] Burch, J., E. Clarke, K. McMillan, D. Dill and L. Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, Information and Computation **98** (1992), pp. 142–170.

[11] Cleaveland, R. and B. Steffen, *A Linear–Time Model Checking Algorithm for Alternation–Free* μ*–Calculus*, Formal Methods in System Design **2** (1993), pp. 121–147.

[12] Cook, S., *The Complexity of Theorem Proving Procedures*, in: *Theory of Computing*, 1971, ACM symposium proceedings.

[13] Drechsler, R., B. Becker and A. Jahnke, *On Variable Ordering and Decomposition Type Choice in OKFDDs*, in: W. Grass and M. Mutz, editors, *Anwendung formaler Methoden beim Entwurf von Hardwaresystemen* (1995), pp. 98–107, GI/ITG workshop proceedings.

[14] Emerson, E. and C.–L. Lei, *Temporal Reasoning under Generalised Fairness Constraints*, in: B. Monien and G. Vidal–Naquet, editors, *Theoretical Aspects of Computer Science* (1986), pp. 21–36, proceedings TACAS '86 conference.

[15] Gasser, R., "Efficiently Harnessing Computational Resources for Exhaustive Search," Ph.D. thesis, computer science department, Eidgenössische Technische Hochschule Zürich (1994).

[16] Kozen, D., *Results on the Propositional μ–Calculus*, Theoretical Computer Science **27** (1983), pp. 333–354.

[17] Sanghavi, J., R. Ranjan, R. Brayton and A. Sangiovanni–Vincentelli, *High–Performance BDD Package by Exploiting Memory Hierarchy*, in: *Design Automation*, 1996, pp. 635–640, proceedings DAC '96.

[18] Schaeffer, J., R. Lake, P. Lu and M. Bryant, *Chinook: The Man–Machine World Checkers Champion*, AI Magazine **17** (1996), pp. 21–29.

[19] Sentovich, E., *A Brief Study of BDD Package Performance*, in: M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design*, LNCS 1166 (1996), pp. 389–403, proceedings FMCAD '96 conference.

[20] Shilov, N. and K. Yi, *Puzzles for Learning Model Checking, Model Checking for Programming Puzzles, Puzzles for Learning Model Checkers*, in: H. Bowman, editor, *Formal Methods Elsewhere*, Electronic Notes in Theoretical Computer Science (2001), proceedings FM–Elsewhere 2001 workshop.

[21] Tarski, A., *A Lattice–Theoretical Fixpoint Theorem and its Applications*, Pacific Journal of Mathematics **5** (1955), pp. 285–309.

[22] University of Alberta, Department of Computer Science, *Chinook Website* (May 2000), `http://www.cs.ualberta.ca/∼chinook`.