A Tutorial Introduction to Symbolic Model Checking

Chapter · January 2003

DOI: 10.1007/0-306-48088-3_5

1 author:

David Déharbe
ClearSy System Engineering
113 PUBLICATIONS **692** CITATIONS

SEE PROFILE

A tutorial introduction to symbolic model checking

David Déharbe
Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Campus Universitário — Lagoa Nova
59078-970 Natal, RN, Brazil
david@dimap.ufrn.br

Abstract

CTL model checking is a fully automatic formal verification technique that can be used to prove important classes properties on finite-state transition systems. Combined with a graph-based representation of propositional logic named binary decision diagrams (BDDs), CTL model checking has been successfully applied to check a variety of industrial-size concurrent systems. This paper gives a tutorial presentation of CTL model checking and its symbolic BDD-based version implementation.

1 Introduction

Model checking is a decision procedure to check that a given structure is a model of a given formula. [11] and [29] presented independently a fully automatic model checking algorithm for the branching time temporal logic CTL in finite-state transition systems. The verification was performed as a graph traversal, linear in both the size of the formula and in the size of the model. This algorithm has been used to verify systems of up to several million states and transitions, which is enough in practice only for small systems.

A major breakthrough has been achieved when [25] proposed to represent the state-transition graph and implement efficiently the search algorithms using binary decision diagrams [6] (BDDs), which are a concise representation for the propositional logic. In this algorithm, called symbolic model checking, the BDDs are used to represent and operate on the characteristic functions of both sets of transitions and sets of states of the graph. Since sets are not explicitly enumerated, but represented by their characteristic functions, the size of the verified model is not bound by the memory of the computer carrying the verification. This turns possible the verification of systems that are several orders of magnitude larger than was previously achieved.

Outline: We present these techniques by first discussing Kripke structures, the state-transition model of preference in the literature on the subject (Section 2). We then introduce the temporal logic CTL and a corresponding decision procedure, temporal logic model checking (Section 3). We present the connection between propositional logic and Kripke structures, the binary decision diagram representation for propositional logic formulas, and symbolic model checking algorithms (Section 4). Finally, we conclude with a discussion on some of the ongoing research on the subject (Section 5).

2 Kripke structures

2.1 Definitions

Among the numerous concurrency models proposed in the last 20 years [33], Kripke structures are one of the most commonly used in the scope of temporal logic model checking. Kripke structures are interleaving, non-deterministic, state-based models and suitable to represent a wide-range of practical problems: hardware, protocols, software, to name some of the most important.

Informally, Kripke structures are finite-state transition systems, where the labelling is associated with states, instead of the more traditional labelling of transitions.

Definition 1 (Kripke structure) *Let P be a finite set of boolean propositions. A Kripke structure over P is a quadruple $M = (S, T, I, L)$ where:*

- *S is a set of states (when S is finite, we say that M is a* finite Kripke structure*);*
- *$T \subseteq S \times S$ is a transition relation, such that $\forall s \in S \bullet \exists s' \in S, (s, s') \in T$;*
- *$I \subseteq S$ is the set of initial states;*
- *$L : S \to 2^P$ is a labeling function.*

The labelling function L associates each state with a set of boolean propositions true in that state.

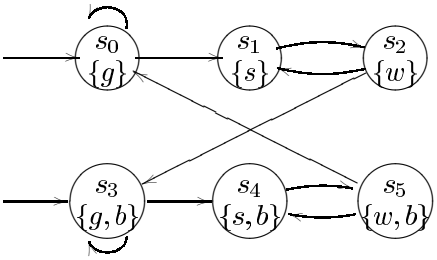
Example 1 (Kripke structure) *The Kripke structure ABPsender describes the sender component in the alternating bit protocol. The set of atomic propositions is $P = \{g, s, w, b\}$, and $ABPsender = (S_A, T_A, I_A, L_A)$ where:*

- *the set of states is such that $S_A = \{s_0, s_1, s_2, s_3, s_4, s_5\}$;*
- *the transition relation is such that*

$$T_A = \{(s_0, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_1), (s_2, s_3), ; \\ (s_3, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_4), (s_5, s_0)\}$$

- *the set of initial states is such that $I_A = \{s_0, s_3\}$;*

Figure 1 State transition diagram for Kripke structure *ABPsender*



- *the labeling function is such that* $L_A = \{s_0 \mapsto \{g\}, s_1 \mapsto \{s\}, s_2 \mapsto \{w\}, s_3 \mapsto \{g,b\}, s_4 \mapsto \{s,b\}, s_5 \mapsto \{w,b\}\}$.

The state transition diagram of ABPsender is displayed in Figure 1.

In this protocol, the sender tags messages with a control bit alternatively set high and low. The receiver shall acknowledge each message, attaching the corresponding control value. The sender waits for the acknowledgement and checks that the control value is correct before sending another data message. If necessary, the same data is sent again. In this Kripke structure, b indicates the state of the control bit. g is set when the sender is getting the data from the user. s states that the sender is sending the message to the transmission medium. Finally, w indicates that the sender is waiting for the acknowledgement. The data messages are not modeled in this version of ABPsender.

A path π in the Kripke structure M is a possibly infinite sequence of states $(s_1, s_2, \ldots)$ such that $\forall i \bullet i \geq 1, (s_i, s_{i+1}) \in T$. $\pi(i)$ is the i^{th} state of π. The set of states reachable from I, denoted RS, is the set of states s such that there exists a path to this state:

$$RS = \{s \in S \mid \exists \pi, ((\pi(1) \in I) \wedge \exists i \geq 1, (\pi(i) = s))\} \tag{1}$$

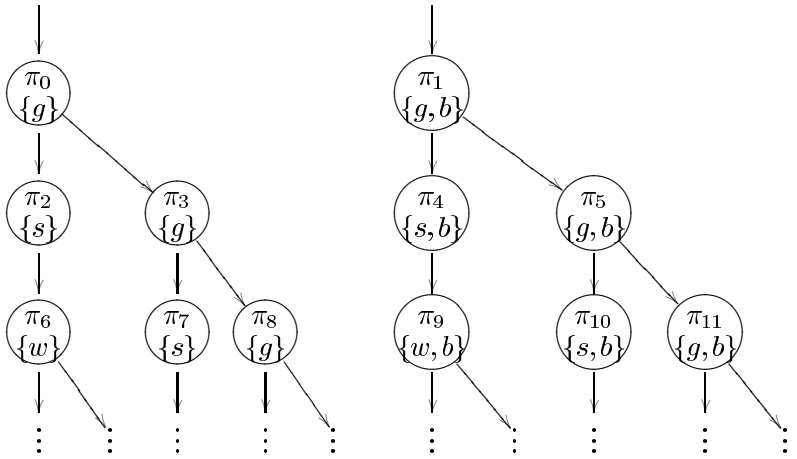
2.2 Computation tree

The computation tree of a Kripke structure is obtained by an operation similar to that of unfolding in CCS [27]. Consequently, computation trees form a special class of Kripke structures, such that the transition relation is acyclic, and only branches away from the initial states.

Definition 2 (Computation tree) *Let P be a finite set of boolean propositions. A computation tree is a Kripke structure $M = (S, T, I, L)$ over P, such that:*

- *every state is reachable;*
- *Let T^+ the transitive closure of T. $\forall s \in S \bullet (s, s) \notin T^+$;*

Figure 2 Partial state transition diagram of $ct(ABPsender)$



- $\forall s, s', s'' \in S \bullet (s', s) \in T \wedge (s'', s) \in T \Rightarrow s' = s''$.

For any Kripke structure M, it is possible to associate a computation tree M' such that the sets of states of M' is isomorphic to the set of the finite paths of M.

Definition 3 *Let $M = (S, T, I, L)$ be a Kripke structure. The computation tree of M, denoted $ct(M)$, is the Kripke structure (S', T', I', L') such that:*

- *S' consists of all finite paths of M that start at initial states;*
- *$(\pi, \pi') \in T'$ iff $\pi = (s_1, \ldots s_n)$, $\pi' = (s_1, \ldots, s_n, s_{n+1})$ and $(s_n, s_{n+1}) \in T$.*
- *I' consists of all paths of M with only one (initial) state;*
- *For any path $\pi = (s_1, \ldots s_n)$ of M, $L'(\pi) = L(s_n)$.*

Example 2 (Computation tree) *Figure 2 depicts the initial part of the state transition diagram of the infinite Kripke structure $ct(ABPsender)$. For instance, state π_9 in $ct(ABPsender)$ corresponds to path $s_3 s_4 s_5$ in $ABPsender$. Note how the labeling function is preserved between the origin and destination of the transitions in the original Kripke structure and the corresponding computation tree.*

3 Temporal Logic Model Checking

3.1 Computation tree logic

Computation Tree Logic (CTL for short) is a logic to reason about Kripke structures and is interpreted over their computation tree. Within the framework of

modal and temporal logics, CTL can be classified as a branching-time, discrete temporal logic.

3.1.1 Syntax

The set $T_{\mathsf{CTL}}(P)$ of Computation Tree Logic (CTL for short) formulas over a set of propositions P is the smallest set such that $P \subseteq T_{\mathsf{CTL}}(P)$ and, if f and g are in $T_{\mathsf{CTL}}(P)$, then $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{AX}f$, $\mathbf{EG}f$, $\mathbf{AG}f$, $\mathbf{EF}f$, $\mathbf{AF}f$, $\mathbf{E}[f\mathbf{U}g]$, $\mathbf{A}[f\mathbf{U}g]$ are in $T_{\mathsf{CTL}}(P)$.

Each temporal logic operator is composed of:

- a path quantifier: **E**, for some path, or **A**, for all paths;
- followed by a state quantifier: **E**, next state on the path, **U**, until, **G**, globally, or **F**, eventually.

3.1.2 Semantics

The semantics of CTL are defined with respect to a Kripke structure $M = (S, T, I, L)$ over a set of atomic propositions P. If f is in $T_{\mathsf{CTL}}(P)$, $M, s \models f$ means that f holds at state s of M.

Let f and g be in $T_{\mathsf{CTL}}(P)$, then

1. $M, s \models p$ iff $p \in L(s)$.
2. $M, s \models \neg f$ iff $M, s \not\models f$.
3. $M, s \models f \wedge g$ iff $M, s \models f$ and $M, s \models g$.
4. $M, s \models \mathbf{EX}f$ iff there exists a state s' of M such that $(s, s') \in T$ and $s' \models f$. I.e., s has a successor where f is valid.
5. $M, s \models \mathbf{EG}f$ iff there exists a path π of M such that $\pi(1) = s$ and $\forall i \geq 1 \bullet M, \pi(i) \models f$. I.e., s is at the start of a path where f holds globally.
6. $M, s \models \mathbf{E}[f\mathbf{U}g]$ iff there exists a path π of M such that $\pi(1) = s$ and $\exists i \geq 1 \bullet (M, \pi(i) \models g \wedge \forall j, i > j \geq 1 \bullet M, \pi(j) \models f)$. I.e., s is at the start of a path where g holds eventually and f holds until g becomes valid (nothing is said about f when g turns valid).

The other temporal logic operators can be defined in terms of **EX**, **EG** and **E**[**U**]:

$$\begin{aligned}
\mathbf{AX}f &= \neg\mathbf{EX}\neg f \\
\mathbf{AG}f &= \neg\mathbf{EF}\neg f \\
\mathbf{AF}f &= \neg\mathbf{EG}\neg f \\
\mathbf{EF}f &= \mathbf{E}[\mathit{true}\mathbf{U}f] \\
\mathbf{A}[f\mathbf{U}g] &= \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \wedge \neg\mathbf{EG}\neg g
\end{aligned}$$

Figure 3 Illustration of universal CTL operators

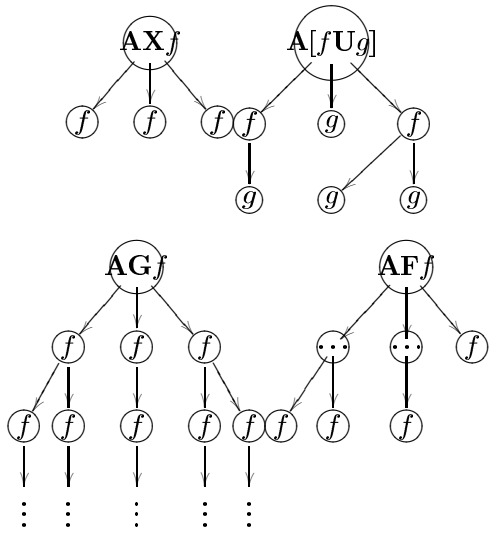


Figure 3 pictures the semantics of the four universal temporal operators (vertical dots indicate paths where f holds infinitely).

Definition 4 *A formula f is valid in structure M if it is valid for all initial states:*

$$M \models f \text{ iff } \forall s \in I \bullet M, s \models f.$$

Example 3 (CTL formulas) *The following CTL formulas state properties of the ABPsender Kripke structure:*

- **AG**$(s \vee w \vee g)$*: The sender is always in one of the three states: getting from the sender, sending a message, or waiting for an acknowledgement.*
- **EG**$(\neg s \wedge \neg w)$*: There is an execution path of the sender such that it is never sending or waiting.*

3.2 Algorithm

Given a Kripke structure $M = (S, T, I, L)$ over a set of propositions P and a CTL formula f, its specification, the algorithm given in [12] repeatedly labels the states of M with the sub-formulas of f, starting with the sub-formulas of length 1 (i.e. atomic propositions in P) and finishing with f itself.

Figure 4 sketches this algorithm. Function *Emc* first calls the graph labeling function with the given formula and then checks that all initial states are labeled. Labelling function *Label* recurses down the structure of the formula (function

Figure 4 Labeling algorithm for CTL formulas

```
function Emc(M: Kripke structure, f: CTL) : boolean
begin
    Label(M, f)
    for each s ∈ I do
        if f ∉ L(s) then return false
    return true
end function Emc

function Label(M: Kripke structure, f: CTL) : void
begin
    if f ∈ P then return;
    for each f_i ∈ args(f) do
        Label(M, f_i)
    case f of
        when ¬f_1: for each s ∈ S do
                   if f_1 ∉ L(s) then L(s) ← L(s) ∪ {f}
        when f_1 ∧ f_2: for each s ∈ S do
                   if f_1 ∈ L(s) ∧ f_2 ∈ L(s) then L(s) ← L(s) ∪ {f}
        when EX f_1: LabelEX(M, f_1)
        when EG f_1: LabelEG(M, f_1)
        when E[f_1 U f_2]: LabelEU(M, f_1, f_2)
    end case
end function Label
```

args, given a CTL formula f, returns the set of sub-formulas of f). The terminal case is when the formula is an atomic function. If the formula is a boolean function, each state is labeled according to the previous labeling of the function arguments. Finally, to deal with temporal operators, special-purpose labeling functions are invoked.

Function *LabelEX* deals with $\mathbf{EX}f$ formulas and is given in Figure 5. For each transition $t = (s_1, s_2)$, if s_2 is labeled with f, s_1 has a successor where f is valid, and formula $\mathbf{EX}f$ is added to the label of the s_1.

Figure 5 Labeling algorithm for $\mathbf{EX}f$ formulas

```
function LabelEX(M: Kripke structure, f: CTL) : void
begin
    for each t = (s_1, s_2) ∈ T do
        if f ∈ L(s_2) then L(s_1) ← L(s_1) ∪ {EX f}
end function LabelEX
```

Function *LabelEU* handles $\mathbf{E}[f\mathbf{U}g]$ formulas (Figure 6). $\mathbf{E}[f\mathbf{U}g]$ is valid in

a state if and only if there is a finite path starting in this state, where f is always valid but in the last state, where g is valid. First, each state already with g is also labeled with $\mathbf{E}[f\mathbf{U}g]$. Then, function *LabelEUaux* is invoked and backtracks along the transitions while f appears in the labels of the states. Each state found along such paths is labeled with formula $\mathbf{E}[f\mathbf{U}g]$. To avoid infinite loops, this backtracking also stops as soon as it meets a state already labeled with $\mathbf{E}[f\mathbf{U}g]$.

Figure 6 Labeling algorithm for $\mathbf{E}[f\mathbf{U}g]$ formulas

```
function LabelEU(M: Kripke structure, f: CTL, g: CTL) : void
begin
    for each s ∈ S do
        if g ∈ L(s) then
            L(s) ← {E[fUg]} ∪ L(s)
            for each t = (s', s) ∈ T do
                LabelEUaux(M, f, g, s')
end function LabelEU

function LabelEUaux(M: Kripke structure, f: CTL, g: CTL, s: state) : void
begin
    if f ∈ L(s) ∧ E[fUg] ∉ L(s) then
        L(s) ← L(s) ∪ {E[fUg]}
        for each (s', s) ∈ T do
            LabelEUaux(M, f, s')
end function LabelEUaux
```

Finally, function *LabelEG* handles $\mathbf{EG}f$ formulas (Figure 7). $\mathbf{EG}f$ is valid in a state s if, and only if, there is an infinite path, starting at s and where f holds on each state. To detect such situations, it is necessary to find cycles in the transition graph along which f is always valid. This is the role of auxiliary routine *LabelEGaux*. *LabelEGaux* has an additional parameter s, which is the state currently visited, and returns a boolean to indicate if $\mathbf{EG}f$ is valid in s. Additionally, two flags are associated with each state: $checked(s)$ and $mark(s)$. $checked(s)$ indicates if the algorithm has already computed if $\mathbf{EG}f$ is valid or not in state s. $mark(s)$ is true if the algorithm has not yet checked if $\mathbf{EG}f$ holds in s, and if s starts a finite path along which f always hold. *LabelEGaux* does a depth-first search along the transition graph as long as:

1. It does not reach an already checked state s. If it does, it stops backtracking and returns a boolean to tell if $\mathbf{EG}f$ is valid in s or not.
2. It does not reach a state that is marked. If it does, then it has found a cycle where f is always valid. In this case it returns *true*.
3. It does not reach a state s where f does not hold. If it does, then the value returned is $false$ (this is implicit in this algorithm).

4. Otherwise f is valid in s and $\mathbf{EG}f$ is potentially valid in s. The algorithm then calls itself recursively and checks if $\mathbf{EG}f$ is valid in one of the successors of f. As soon as one such successor is found, then formula $\mathbf{EG}f$ is added to the label set of s and the algorithm returns immediately *true*. If no such successor is found, then formula $\mathbf{EG}f$ is not added to the label set and the algorithm returns $false$.

Figure 7 Labeling algorithm for $\mathbf{EG}f$ formulas

```
function LabelEG(M: Kripke structure, f: CTL) : void
begin
    for each s ∈ S do
        checked(s) ← false, mark(s) ← false
    for each s ∈ S do
        LabelEGaux(M, f, s)
end function LabelEG

function LabelEGaux(M: Kripke structure, f: CTL, s: state) : boolean is
begin
    if ¬checked(s) then
        if mark(s) then
            return true
        if f ∈ L(s) then
            mark(s) ← true
            for each (s, s') ∈ T do
                if LabelEGaux(M, s', f) then
                    L(s) ← L(s) ∪ {EGf}
                    checked(s) ← true
                    return true
            mark(s) ← false
        checked(s) ← true
    return (EGf ∈ L(s))
end function LabelEGaux
```

Example 4 (Verification of *ABPsender***)** *To illustrate the model checking algorithm, we apply it to the verification of the ABPsender (Figure 1). More specifically, we check that formula* $\mathbf{EG}(\neg s \wedge \neg w)$ *is verified by ABPsender, which is computed with the function call* $Emc(ABPsender, \mathbf{EG}(\neg s \wedge \neg w))$.

The first step of the algorithm Emc (Figure 4) consists in labeling recursively the structure with the formulas and its sub-formulas. This is done by invoking algorithm Label with parameters ABPsender and $\mathbf{EG}(\neg s \wedge \neg w)$. *Algorithm Label first labels the states of the graph with each one of the sub-formulas of the specification that are valid in those states. Since these sub-formulas are all*

boolean, the application of Label is trivial and yields the following state labeling:

$$L(s_0)=\{g, \neg s, \neg w, \neg s \wedge \neg w\}$$
$$L(s_1)=\{w, \neg s\}$$
$$L(s_2)=\{s, \neg w\}$$
$$L(s_3)=\{g, b, \neg s, \neg w, \neg s \wedge \neg w\}$$
$$L(s_4)=\{w, b, \neg s\}$$
$$L(s_5)=\{s, b, \neg w\}$$

Next, function Label invokes LabelEG(ABPsender, $\neg s \wedge \neg w$*) (Figure 7). The flags checked and mark of each state are initialized to false. Then function LabelEGaux is invoked on each state. We suppose that the first state to be inspected is* s_0 *and we trace the corresponding call LabelEGaux(ABPsender,* $\neg g \wedge \neg w, s_0$*). Since* s_0 *is not yet checked and formula* $\neg g \wedge \neg w$ *belongs to* $L(s_0)$*, the mark flag of* s_0 *is assigned true and for each transition leaving* s_0*, function LabelEGaux is called on the destination. Suppose transition* (s_0, s_0) *is chosen first. Then LabelEGaux is invoked again on state* s_0*, tests the flag mark, which is now set, and returns true. The execution flows continues from the first invocation of LabelEGaux and adds formula* $\mathbf{EG}\neg s \wedge \neg w$ *to set* $L(s_0)$*, sets true the checked flag and returns true. This operation is repeated for each state of the structure. When LabelEG returns, the state labeling of ABPsender is:*

$$L(s_0)=\{g, \neg s, \neg w, \neg s \wedge \neg w, \mathbf{EG}(\neg s \wedge \neg w)\}$$
$$L(s_1)=\{w, \neg s\}$$
$$L(s_2)=\{s, \neg w\}$$
$$L(s_3)=\{g, b, \neg s, \neg w, \neg s \wedge \neg w, \mathbf{EG}(\neg s \wedge \neg w)\}$$
$$L(s_4)=\{w, b, \neg s\}$$
$$L(s_5)=\{s, b, \neg w\}$$

The second part of Emc executes then and checks that for each initial state of ABPsender the formula belongs to the label set, which is the case. Therefore $ABPsender \models \mathbf{EG}(\neg s \wedge \neg w)$.

3.3 Experimental results and Conclusion

To illustrate the original model checking algorithm, a fully-functional implementation was demonstrated with a version of the complete alternating bit protocol that had a total of 251 states, with running times taking about 10 seconds for each formula to be verified [12]. After further optimizations, a parallel version of the model checking algorithm, implemented on a vector architecture, was able to verify a Kripke structure with 131,072 states and 67,108,864 transitions, its specification being a CTL formula with 113 sub-formulas. The time reported for this experiment was 225 seconds.

Despite these somehow impressive sounding results, in practice, the model checking presented above is not efficient enough to deal with industrial designs.

In concurrent systems, the size of the state space grows exponentially with the number of components. For instance, the model of a sequential circuit with n flip-flops is a Kripke structure with potentially 2^n states: for $n = 32$, the order of magnitude of the number of potential states is 10^9. This phenomenon is known as the *state space explosion*, makes it practically impossible to represent exhaustively the set of states and the set of transitions of most systems.

4 Symbolic Model Checking

One possible way to avoid (or, at least, to delay) the state space explosion is to represent sets of states and transitions by their characteristic function rather than by enumeration. It is the purpose of Section 4.1 to explain how propositional logic may be used as a language to define and manipulate the characteristic functions. In Section 4.2, we present binary decision diagrams (BDDs), an efficient graph-based implementation of propositional logic. Finally, Section 4.3 details the symbolic version of the model checking algorithms presented in the previous section.

4.1 Kripke structures and propositional logic

4.1.1 Representing states and transitions

Let $M = (S, T, I, L)$ be a Kripke structure over $P = \{v_1, \ldots, v_n\}$. Let $\mathbf{v}$ denote $(v_1, \ldots v_n)$. The characteristic function of a state $s \in S$, denoted $[s]$, is defined as:

$$[s](\mathbf{v}) = \left(\left(\bigwedge_{v_i \in L(s)} v_i \right) \wedge \left(\bigwedge_{v_i \notin L(s)} \neg v_i \right) \right)$$

The definition of the characteristic function is extended to sets of states with the following definitions:

$$[\{\}](\mathbf{v}) = false$$
$$[\{x\} \cup X](\mathbf{v}) = [x](\mathbf{v}) \vee [X](\mathbf{v})$$

Let $P' = \{v'_1, \ldots v'_n\}$ be a set of fresh boolean propositions. The characteristic function of a transition $t = (s_1, s_2) \in T$, denoted $[t]$, is defined as:

$$[t](\mathbf{v}, \mathbf{v}') = [s_1](\mathbf{v}) \wedge [s_2](\mathbf{v}')$$

This definition can be extended to represent sets of transitions as for sets of states.

Example 5 (Characteristic function) *In the Kripke structure ABPsender, the characteristic functions of the initial state s_0, of the transition (s_0, s_1) and*

of the initial states I are, respectively:

$$\begin{aligned}
[s_0] &= g \wedge \neg s \wedge \neg w \wedge \neg b \\
[(s_0, s_1)] &= (g \wedge \neg s \wedge \neg w \wedge \neg b) \wedge (\neg g' \wedge s' \wedge \neg w' \wedge \neg b') \\
[I] &= (g \wedge \neg s \wedge \neg w \wedge \neg b) \vee (g \wedge \neg s \wedge \neg w \wedge b) \\
&= (g \wedge \neg s \wedge \neg w) \\
[T] &= (b \underline{\vee} b') \wedge \neg g \wedge \neg s \wedge w \wedge g' \wedge \neg s' \wedge w' \\
&\quad \vee (b \leftrightarrow b') \wedge \quad (g \wedge \neg s \wedge \neg w \wedge \neg w' \wedge (g' \underline{\vee} s')) \\
&\qquad\qquad \vee (\neg g \wedge s \wedge \neg w \wedge \neg g' \wedge \neg s' \wedge w') \\
&\qquad\qquad \vee (\neg s \wedge \neg g' \wedge s' \wedge \neg w' \wedge (g \underline{\vee} w))
\end{aligned}$$

To simplify notations, in the rest of the paper we will identify $[X]$ with X.

4.1.2 State space traversal

Let $M = (S, T, I, L)$ be a Kripke structure over P. The image of a set of states $X \subseteq S$ is the set of states that can be reached in one transition from X:

$$\{s \in S \,|\, \exists s' \in X, (s', s) \in T\}$$

The characteristic function of the image of X, denoted $Forward(M, X)$, is:

$$Forward(M, X)(\mathbf{v}') = \exists \mathbf{v}, X(\mathbf{v}) \wedge T(\mathbf{v}, \mathbf{v}') \tag{2}$$

Conversely, the inverse image of a set of states $X \subseteq S$, is the set of states from which X can be reached in one transition:

$$\{s \in S \,|\, \exists s' \in X, (s, s') \in T\}$$

The characteristic function of the inverse image of a set of states X, denoted $Backward(M, X)$, is:

$$Backward(M, X)(\mathbf{v}) = \exists \mathbf{v}', X(\mathbf{v}') \wedge T(\mathbf{v}, \mathbf{v}') \tag{3}$$

4.2 Binary decision diagrams

Binary Decision Diagrams (BDDs for short) form a heuristically efficient data structure to represent formulas of the propositional logic. Let P be a totally ordered finite set of boolean propositions. Let f be a boolean formula over P, $bdd(f)$ is the BDD representing f, and $|bdd(f)|$ is the size of this BDD. [6] showed that BDDs are a canonical representation: two equivalent formulas are represented with the same BDD:

$$f \Leftrightarrow g \text{ iff } bdd(f) = bdd(g)$$

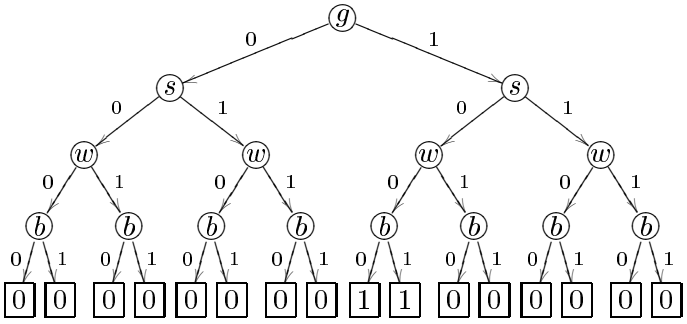
Moreover, most boolean operations can be performed efficiently with BDDs. Let $|b|$ denote the size of BDD b:

- $bdd(\neg f)$ is computed in constant time $O(1)$.
- $bdd(f \vee g)$ is realized in $O(|bdd(f)|.|bdd(g)|)$.
- $bdd(\exists x, f)$ is performed in $O(|bdd(f)|^2)$.

In this paper, we will use usual boolean operators to denote the corresponding operation on BDDs, e.g. $bdd(f) \vee bdd(g) = bdd(f \vee g)$.

We explain the basic principles of the BDD representation on an example. Fig. 8 presents the binary decision tree for the reachable states of Kripke structure *ABPsender*: $g \wedge \neg s \wedge \neg w$. The binary tree representation of a formula is exponential in the number of free boolean propositions in the formula.

Figure 8 Binary tree for $g \wedge \neg s \wedge \neg w$.

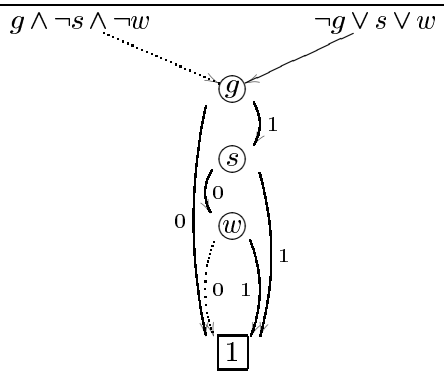


The corresponding BDD is obtained by repeatedly applying the following rules:

- remove duplicate terminal vertices, i.e. after this operation there should be only two leafs, labeled respectively with 0 and 1;
- remove duplicate vertices bottom-up (two vertices are duplicate when they have the same label and their children are identical),
- remove opposite vertices (two vertices are opposite when they are labeled with the same variable, and they have opposite children),
- remove redundant tests (a node is a redundant test if its children are identical).

Fig. 9 presents the BDD of the characteristic function for the reachable states of Kripke structure *ABPsender*, with variable ordering $g < s < w < b$. Dotted edges indicate that the function on the target node shall be negated. Therefore, the same BDD node is used to represent both a function and its negation (in Figure 9, the BDD represents $\neg g \vee s \vee w$ as well), it is interpreted differently according the type of the edge that is pointing to it. With variable ordering $g < g' < s < s' < w < w' < b < b'$, the BDD for the transition relation has 22 nodes.

Figure 9 BDD for $g \wedge \neg s \wedge \neg w$.



[6] showed that some functions have an exponential BDD representation for any variable ordering, and that finding the optimum variable ordering is NP-hard. However, in practice, heuristic methods generally achieve a good variable ordering, when such ordering exists.

In a Kripke structure, states, transitions and sets thereof can be characterized with propositional logic formulas. These formulas can be represented and manipulated via their BDD representation. BDDs proved to be an efficient data structure to perform computations on large Kripke structures.

In the remainder, we will use the following operations on BDs:

- *BddFalse*, *BddTrue* return the BDD for the boolean constants;
- *BddAtom* takes a boolean proposition p as parameter and returns the BDD that represents p;
- *BddNot* takes as parameter the BDD of a boolean formula f and returns the BDD of formula $\neg f$;
- *BddAnd* (resp. *BddOr*) take as parameters the BDDs of two boolean formulas f and g and returns the BDD of $f \wedge g$ (resp. $f \vee g$).
- *BddImplies* is a predicate that takes as parameters the BDDs of two boolean formulas f and g and checks wether f is a logical implication of g.

4.3 Algorithms

4.3.1 Fundamentals of lattices and fixpoints

A lattice is a set with a partial order on the elements of this set, a least element $\bot$, and a greatest element $\top$.

Let P be a non-empty finite set of atomic propositions. Let $M = (S, T, I, L)$ be a finite Kripke structure over P. We consider $(2^S, \subseteq)$ the lattice of subsets of

S with set inclusion as the ordering. The empty set $\{\}$ and S are respectively the least and greatest elements of this lattice. Since a subset of S can be identified with its characteristic function, this lattice can also be interpreted as the lattice of characteristic functions, with boolean implication as ordering, *false* is the least element, and the characteristic function of S is the greatest element.

A function $\tau : 2^S \rightarrow 2^S$ is called a *predicate transformer*. τ is *monotonic* iff $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$. Also τ is $\cup$-*continuous* (resp. $\cap$-*continuous*) when $P_1 \subseteq P_2 \subseteq \ldots$ (resp. $P_1 \supseteq P2 \supseteq \ldots$) implies that $\tau(\cup_i P_i) = \cup_i \tau(P_i)$ (resp. $\tau(\cap_i P_i) = \cap_i \tau(P_i)$). [32] showed that if τ is monotonic, then τ has a least fixpoint, denoted $\mathbf{lfp}Z[\tau(Z)]$, and a greatest fixpoint, denoted $\mathbf{gfp}Z[\tau(Z)]$. Moreover, since the lattice is finite and τ is monotonic, it is also $\cup$-continuous and *cap*-continuous, and:

$$\mathbf{lfp}Z[\tau(Z)] = \cap\{Z \mid \tau(Z) = Z\} = \cup_i \tau^i(\mathit{false}) \tag{4}$$

$$\mathbf{gfp}Z[\tau(Z)] = \cup\{Z \mid \tau(Z) = Z\} = \cap_i \tau^i(\mathit{true}) \tag{5}$$

4.3.2 Fixpoint characterization of CTL operators

CTL symbolic model checking uses the BDD representations of the characteristic functions of sets of states and transitions. The algorithm is based on the fixpoint characterization of the different temporal operators of CTL defined in [11]:

$$\mathbf{EG}f = \mathbf{gfp}Z[f \wedge \mathbf{EX}Z] \tag{6}$$

$$\mathbf{E}[f\mathbf{U}g] = \mathbf{lfp}Z[g \vee (f \wedge \mathbf{EX}Z)] \tag{7}$$

The symbolic model checking algorithm, named *Smc*, is shown in Figure 10, is a predicate that takes as arguments a Kripke structure M and a CTL formula f. It uses the auxiliary routine *SmcAux*, that returns the characteristic function of the states of M that satisfies f, and checks if f is valid in each initial state of M. *SmcAux* itself relies on auxiliary routines *SmcEX* (Figure 11), *SmcEU* (Figure 12) and *SmcEG* (Figure 13). All these routines are used to recurse over the syntactical structure of CTL formulas, and have as arguments M a Kripke structure, f a CTL formula, and as result the BDD of the characteristic function of the set of M states where f is valid.

SmcEX(M, f) computes the states of M where $\mathbf{EX}f$ is valid (Figure 11). It first computes F, the BDD for the characteristic function of the set states of M where f is valid, and returns the inverse image of F. The inverse image is computed with *Backward*, a routine that implements Equation 3.

Similarly, *SmcEU*(M, f, g) computes the states of M where $\mathbf{E}[f\mathbf{U}g]$ is valid (Figure 12). It first computes F and G, characterizing the states of M where f and g are valid, and then computes the least fixpoint defined in Equation 7.

SmcEG(M, f) computes the states of M where $\mathbf{EG}f$ is valid (Figure 13). It first computes F, the BDD for the set of states of M where f is valid, and then computes the greatest fixpoint defined in Equation 6.

Figure 10 Algorithm for symbolic model checking CTL formulas

```
function Smc(M: Kripke structure, f: CTL) : boolean
begin
    return BddImplies(M.I, SmcAux(M, f))
end function Smc
function SmcAux(M: Kripke structure, f: CTL) : BDD
begin
    case f of
        f is a boolean proposition: return BddAtom(f)
        f = ¬f1: return BddNot(SmcAux(M, f1))
        f = f1 ∧ f2: return BddAnd(SmcAux(M, f1), SmcAux(M, f2))
        f = EXf1: return SmcEX(M, f1)
        f = EGf1: return SmcEG(M, f1)
        f = E[f1 U f2]: return SmcEU(M, f1, f2)
end function SmcAux
```

Figure 11 Algorithm for $\mathbf{EX}f$ formulas

```
function SmcEX(M: Kripke structure, f: CTL) : BDD
variables
    F: BDD
begin
    F ←SmcAux(M, f)
    return Backward(M, F)
end function SmcEX
```

Example 6 (Symbolic verification of $ABPsender$**)** *To illustrate the symbolic model checking algorithm, we apply it to the same verification as Example 5: we check that the formula* $\mathbf{EG}(\neg s \wedge \neg w)$ *is verified by* $ABPsender$*, which is computed with the function call* $Smc(ABPsender, \mathbf{EG}(\neg s \wedge \neg w))$ *(Figure 10).*

Most of the computation is carried out by the call $SmcEG(ABPsender, \neg s \wedge \neg w)$ *(Figure 13). Figure 14 contains a trace for the values of the expressions* Q*,* $Backward(ABPsender, Q)$*,* Q' *and* $Q = Q'$ *on the while statement test at the different iterations of the fixpoint computation*[1]*. The result returned by the function call is the BDD for* $g \wedge \neg w \wedge \neg s$ *which is also that of the characteristic function for the set of initial states. Therefore the symbolic model checking returns a true answer, stating that the formula* $\mathbf{EG}\neg s \wedge \neg w$ *is valid in the Kripke structure* $ABPsender$*.*

[1] Actually the values displayed in this table are that of the boolean formulas represented by Q and Q', instead of the less human-friendly BDDs.

Figure 12 Algorithm for $\mathbf{E}[f\mathbf{U}g]$ formulas

```
function SmcEU(M: Kripke structure, f: CTL, g: CTL) : BDD
variables
    Q, Q', F, G: BDD
begin
    F ← SmcAux(M, f)
    G ← SmcAux(M, g)
    Q ← BddFalse
    Q' ← BddOr(G, BddAnd(F, Backward(M, Q)))
    while Q ≠ Q' do
        Q ← Q'
        Q' ← BddOr(G, BddAnd(F, Backward(M, Q)))
    end while
    return Q
end function SmcEU
```

4.3.3 Results and extensions to symbolic model checking

Symbolic model checking has been used to verify a large variety of systems: hardware descriptions [19], software [2], protocols [26, 13]. The size of the Kripke structures used in these verification has been routinely much larger than 10^{20} [8].

An extremely useful feature of model checking is the possibility to compute counterexamples (or witnesses) when a universal formula is false (when an existential formula is true) [15]. For instance, the counterexample of an $\mathbf{AG}f$ formula is a path from an initial state to a state where f is not valid.

In practice, symbolic model checking is well-suited for the verification of the control components of a system. However it performs poorly with data parts. The reason is that BDDs are ill-suited to represent arithmetic expressions or other data-intensive operations. Practically this means that symbolic model checking cannot be used to uncover bugs such as the one found in the Pentium chip floating point division unit. An approach to verify this type of systems has been to combine model checking using other data structures than BDDs to represent the data parts of the system under verification. Word-level model checking [17] is an example of such approach. Word-level model checking uses functions mapping boolean vectors into the integers to model the system under verification. The internal representation of these functions is a combination of two different classes of data structures: multi-terminal binary decision diagrams (MTBDD) represent the control parts and binary moment diagrams [7] (BMD) represent the data parts.

Another limitation of symbolic model checking lies in the expressiveness of the specification logic CTL. Properties asserted in CTL are of a qualitative nature, for instance *if A happens then necessarily B happens in the future*. To express quantitative properties, such as *if A then necessarily B will happen be-*

Figure 13 Algorithm for **EG**f formulas

```
function SmcEG(M: Kripke structure, f: CTL) : BDD
variables
    Q, Q', F: BDD
begin
    F ← SmcAux(M, f)
    Q ← BddTrue
    Q' ← BddAnd(F, Backward(M, Q))
    while Q ≠ Q' do
        Q ← Q'
        Q' ← BddAnd(F, Backward(M, Q))
    end while
    return Q
end function SmcEG
```

Figure 14 Trace of function call $SmcEG(ABPsender, \mathbf{EG}\neg s \wedge \neg w)$

	F	Q	$Backward(Q)$	Q'	$Q = Q'$
Init.	$\neg s \wedge \neg w$	$true$	$(\neg g \wedge w \wedge \neg s) \vee$ $(g \wedge \neg w \wedge \neg s) \vee$ $(\neg g \wedge \neg w \wedge s)$	$g \wedge \neg w \wedge \neg s$	No
Iter. 1	$\neg s \wedge \neg w$	$g \wedge \neg w \wedge \neg s$	$(w \wedge \neg g \wedge \neg s) \vee$ $(g \wedge \neg w \wedge \neg s)$	$g \wedge \neg w \wedge \neg s$	Yes

tween 4 and 8 time units in the future it is necessary to nest several **X** operators into syntactically complex and error prone formulas. One possible solution is to write a preprocessor that converts formulas in a quantitative variant of CTL into an equivalent CTL formula and use the standard symbolic model checking algorithm [20]. Another solution is to develop special-purpose algorithms or model representations for this type of formulas. Some tools [9, 30] have an even more powerful capability of computing the lower and upper bound of all possible intervals between two given events. To consider also continuous-time systems it is necessary to develop completely different techniques based on timed automata [1].

5 Conclusion

We have introduced the main concepts necessary to get the reader familiar with the fundamentals of temporal logic model checking and symbolic model checking. A more advanced treatment of the subject is also presented in [16]. Symbolic model checking is the subject of intensive and diversified researches that all tend to cope with the state-space explosion. These approaches may be classified under the following categories:

Composition: The behavior of a system can be seen as the combination of the behavior of its parts. Therefore one may verify a system based on the verification of its components. The decomposition into elementary proofs and the combination of the results need to be controlled by a compositional proof system. In our case, we would have to combine a compositional proof system for CTL (or a subset thereof) with the symbolic model checking algorithms [21].

Abstraction: The verification of a property needs only consider the aspects of a system that are relevant to this property. Using techniques similar to abstract interpretation, it is possible to directly generate an abstract model of the system being verified on which the verification can take place [14, 23].

Alternative algorithms: More efficient algorithms may be found for some classes of CTL formulas [22, 18]. Also alternatives to the representation of propositional logic with BDDs are now being investigated (e.g. [4]).

BDD improvements: The verification engine basically consists in BDD-based data and operations. Improving BDD management and BDD uses has a direct impact on the performances of symbolic model checkers [31, 35, 34]. Important issues are variable ordering [3], more efficient image and inverse image algorithms [28], and intrinsic implementation issues [10, 5, 24].

References

[1] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for real time systems. In 5^{th} *Symposium on Logic in Computer Science*, pages 414–425, June 1990.

[2] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and R. Reese. Model Checking Large Software Specifications. In 4^{th} *Symposium on the Foundations of Software Engineering*, pages 156–166. ACM/SIGSOFT, Oct. 1996.

[3] A. Aziz, S. Taşiran, and R.K. Brayton. BDD variable ordering for interacting finite state machines. In *31st annual conference on Design Automation conference: DAC'94*, pages 283–288, 1994.

[4] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36th Design Automation Conference: DAC'99*, number 1579 in Lecture Notes in Computer Science. Springer Verlag, 1999.

[5] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEEE Design Automation Conference*, pages 40–45, June 1990.

[6] R.E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions Computers*, C(35):1035–1044, 1986.

[7] R.E. Bryant and Y.-A. Chen. Verification of Arithmetic Circuits with Binary Moments Diagrams. In *32nd ACM/IEEE Design Automation Conference: DAC'95*, pages 535–541, 1995.

[8] J.R. Burch, E.M. Clarke, K.L. Mc Millan, D.L. Dill, and J. Hwang. 10^{20} states and beyond. In *LICS'90: 5^{th} annual IEEE symposium on logic in computer science*, pages 428–439, Philadelphia,PA,Etats-Unis, June 1990. IEEE.

[9] S. Campos and E. Clarke. The Verus language: representing time efficiently with BDDs. In *AMAST Workshop on Real-Time-Systems, Concurrent and Distributed Software*, 1997.

[10] Y.-A. Chen, B. Yang, and R. E. Bryant. Breadth-First with Depth-First BDD Construction: A Hybrid Approach. Technical Report CMU-CS-97-120, Carnegie Mellon University, March 1997.

[11] E.M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

[12] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions On Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[13] E.M. Clarke, O. Grumberg, H. Hirashi, S. Jha, D. Long, and K.L. McMillan. Verification of the Future-Bus+ Cache Coherence Protocol. *Formal Methods in Systems Design*, 6(2):217–232, 1995.

[14] E.M. Clarke, O. Grumberg, and D.E. Long. *19th Annual Symposium on Principles of Programming Languages*, chapter Model checking and abstraction. 1990.

[15] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd ACM/IEEE Design Automation Conference: DAC'95*, 1995.

[16] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Nov. 1999. To be published.

[17] E.M. Clarke, M. Khaira, and X. Zhao. Word-Level Model Checking - Avoiding the Pentium FDIV error. In *33rd ACM/IEEE Design Automation Conference: DAC'96*, pages 645–648, 1996.

[18] D. Déharbe and A. Martins Moreira. Symbolic model checking with fewer fixpoint computations. In *World Congress on Formal Methods: FM'99*, 1999.

[19] D. Déharbe, S. Shankar, and E. Clarke. Model Checking VHDL with CV. In *FMCAD'98: Formal Methods in Circuit Automation Design*, number 1522 in Lecture Notes in Computer Science. Springer Verlag, 1998.

[20] J. Frl, J. Gerlach, and T. Kropf. An efficient algorithm for real-time model checking. In *European Design and Test Conference*, pages 15–21, 1996.

[21] O. Grumberg and D.E. Long. Model Checking and Modular Verification. *ACM Transactions On Programming Languages and Systems*, 16(3):843–871, May 1994.

[22] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *ICCAD'96*, page 82, 1996.

[23] W. Lee, A. Pardo, J.-Y. Jang, G. Hachter, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *International Conference on Computer-Aided Design: ICCAD'96*, page 76, 1996.

[24] D.E. Long. Design of a cache-friendly BDD library. In *ICCAD'98*, pages 639–645, 1998.

[25] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[26] K.L. McMillan and J. Schwalbe. *Shared Memory Multi-Processing*, chapter Formal Verification of the Gigamax Cache Coherency Protocol. MIT Press, 1992.

[27] A.R.G. Milner. *Calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[28] A. Narayan, A.J. Isles, J. Jain, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *1997 IEEE/ACM international conference on Computer-aided design: ICCAD'97*, pages 388–393. 1997.

[29] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Procs. 5^{th} international symposium on programming*, volume 137 of *Lecture Notes in Computer Science*, pages 244–263. Springer Verlag, 1981.

[30] J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In *Advances in Hardware Design and Verification – Proceedings of the International Conference on Correct Hardware and Verification Methods: CHARME'97*, pages 146–163, 1997.

[31] F. Somenzi. *CUDD: CU decision diagrams package – release 2.3.0.* Department of Electrical and Computer Engineering, University of Colorado at Boulder, September 1998.

[32] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, pages 285–309, 1955.

[33] Glynn Winskel and Mogens Nielsen. *Handbook of Logic in Computer Science. Vol. 4: Semantic Modelling*, chapter Models for Concurrency, pages 1–148. Oxford Science Publications, 1995.

[34] B. Yang. *Optimizing Model Checking based on BDD Characterization.* PhD thesis, School of Computer Science – Carnegie Mellon University, May 1999. Available as research report CMU-CS-99-129.

[35] B. Yang, R.E. Bryant, D.R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R.K. Ranjan, and F. Somenzi. A Performance Study of BDD-Based Model Checking. In *Formal Methods in Computer-Aided Design: FMCAD'98*, number 1522 in Lecture Notes in Computer Science. Springer Verlag, 1998.