Symbolic Model Checking with Fewer Fixpoint Computations

David Déharbe* and Anamaria Martins Moreira

Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Campus Universitário — Lagoa Nova
59072-970 Natal, RN, Brazil
{david,anamaria}@dimap.ufrn.br
http://www.dimap.ufrn.br/∼david,∼anamaria

Abstract. Symbolic model checking, *smc*, is a decision procedure that verifies that some finite-state structure is a model for a formula of Computation Tree Logic (CTL). *smc* is based on fixpoint computations. Unfortunately, as the size of a structure grows exponentially with the number of state components, *smc* is not always powerful enough to handle realistic problems.
We first show that a subset of CTL formulas can be checked by testing simple sufficient conditions, that do not require any fixpoint computation. Based on these observations, we identify a second, larger, subset of CTL that can by verified with fewer fixpoint computations than *smc*. We propose a model checking algorithm for CTL that tests the identified sufficient conditions whenever possible and falls back to *smc* otherwise. In the best (resp. worst) case, the complexity of this algorithm is exponentially better (resp. the same) in terms of state components than that of *smc*.

1 Introduction

Model checking is an algorithm for computing the truth of a formula expressed in some logic in a given model. [4] and [15] presented independently a fully automatic model checking algorithm for the branching time temporal logic CTL in finite-state transition systems, linear in 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 [5], which is enough in practice only for small systems.

A big step forward was made when [14] proposed a new model checking algorithm for CTL, based on fixpoint computations of sets of states. In this algorithm, called symbolic model checking, binary decision diagrams [3] are used to represent both the transitions and the states of the model. Since sets of states are represented in intention by their characteristic functions, the size of the verified

* The research presented in this paper has been partially financed by CNPq.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 272–288, 1999.

model is not bound by the memory of the computer running the verification and it is possible to verify systems that have several orders of magnitude more states.

However, most of the systems designed today are much larger and, in order to achieve verification, symbolic model checking must be combined, often manually, with other techniques, such as abstraction and composition [6]. It is important to note that most of the involved techniques are only good in a heuristic sense: although theoretically their computational complexity is extremely large, practically, on many examples, they prove to be efficient [7, 1]. Successful verification of real-world hardware, software and protocols has been achived with these techniques. However, even with these combinations, the verification of large and complex designs often requires too much resources and is either not possible, due to the complexity of the computations involved in the fixpoint computations, or requires human expertise.

In spite of this handicap, usually known as the "state explosion problem", model checking is probably the most successful trend in formal verification today, particularly in the hardware design community. Given this context, research is being carried out with the goal of potentially increasing the number of systems that can be verified through model checking. We can identify three possible ways to attack this problem: reducing the size of the model (e.g. with abstraction [6], symmetry [8], decomposition [12]), reducing the size of the formulas (with a rewrite system) [9], and developping more efficient verification algorithms where the size of the model and of the formula have a smaller impact on the verification activity [11, 2].

The research presented in this paper belongs to the latter. We propose heuristics for verifying an important class of temporal properties (a subset of CTL), with a smaller number of fixpoint computations than the decision procedures that we are aware of. As the traditional algorithm is basically composed of nested fixpoint computations, the complexity of the verification depends directly on the length of these fixpoint computations. We identify a class of properties that may be checked without any fixpoint computations. However, the presented heuristics do not always provide definite results: for a class of formulas, they may give a false positive or a false negative answer. In those identified cases, traditional verification must be carried out. Fortunately, the computational complexity of the heuristics is significantly smaller than those of the traditional algorithm. Experience shows that the performance of the verification process is significantly better when the heuristics works, and that the time and space penalty is unnoticeable to the user when it does not.

These results constitute a generalization of [10], where a similar approach is applied to the class of invariants[1]. Much in the same way, but for a larger class of properties, we present sufficient conditions for these properties to be true in a given model. We also show how the CTL model checking algorithm is to be modified so that it first checks the validity of these conditions before going on with more complex decision procedures.

[1] A property is an invariant of a model if it holds for every reachable state.

Outline: In Sections 2 to 3, we overview the foundations of symbolic model checking: Kripke structures, the class of models considered; and binary decision diagrams, an efficient data structure to represent such structures. We then present in Section 4 syntax and semantics of computation tree logic (CTL) and we define the subsets of CTL which are dealt with by our optimization heuristics. In Section 5, we present the main results of the paper: sufficient conditions for formulas in these subsets to be valid in a given model. We also show, in Section 6, how to incorporate the computation of these sufficient conditions in CTL model checking.

2 Kripke Structures

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 finite set of states.
- $T \subseteq S \times S$ is a transition relation, such that $\forall s \in S, \exists s' \in S, (s, s') \in T$.
- $I \subseteq S$ is the set of initial states.
- $L : S \rightarrow 2^P$ is a labeling function. L is injective and associates with each state a set of boolean propositions true in the state.

A path π in the Kripke structure M is an infinite sequence of states $s_1, s_2, \ldots$ such that $\forall i \geq 1, (s_i, s_{i+1}) \in T$. $\pi(i)$ 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 is a path from an initial state to this state:

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

A property f is an *invariant* of M, if f is true of each state s of RS.

2.1 Characteristic Functions

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.

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

2.2 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:

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

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

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

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:

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

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

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

3 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 if this BDD. [3] 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 [2] $O(1)$.
- $bdd(f \vee g)$ is realized in $O(|bdd(f)| \times |bdd(g)|)$.
- $bdd(\exists x, f)$ 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. 1 presents the binary decision tree for the 2-bit comparison: $x_1 \Leftrightarrow y_1 \wedge x_2 \Leftrightarrow y_2$. The binary tree representation of a formula is exponential in the number of free boolean propositions in the formula.

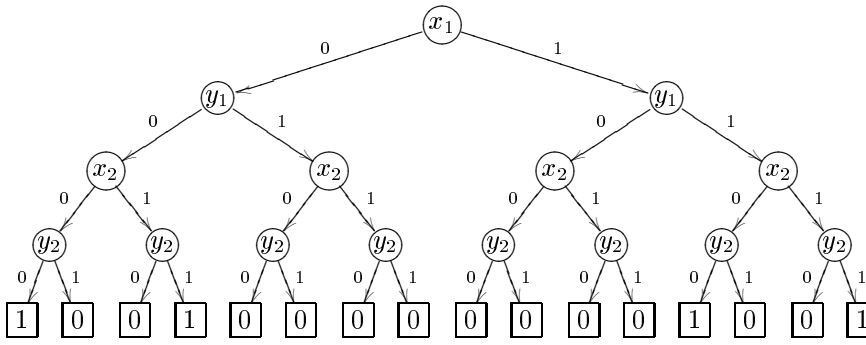

Fig. 1. Binary tree for $x_1 \Leftrightarrow y_1 \wedge x_2 \Leftrightarrow y_2$.

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

- remove duplicate terminal vertices,
- remove duplicate vertices bottom-up,
- remove opposite vertices,
- remove redundant tests.

Fig. 2 presents the BDD of the 2 bit comparator with ordering $x_1 < y_1 < x_2 < y_2$ (dotted edges indicate that the target function shall be negated). For an n-bit comparator, the BDD representation is linear with ordering $x_1 < y_1 < \ldots < x_n < y_n$ and thus is exponentially better than the binary tree representation. However, with ordering $x_1 < \ldots < x_n < y_1 < \ldots < y_n$, the BDD representation is exponential.

[2] Actually, negation on BDDs as presented in [3] is a linear operation. However, [13] proposed a slight modification of the data structure that resulted in reducing the negation to a constant operation. Schematically, the modification consists in tagging edges and identifying the representations of f and $\neg f$. Current BDD packages integrate this modification.

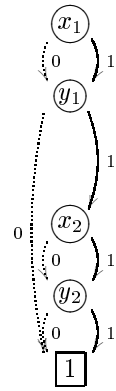

Fig. 2. BDD for $x_1 \Leftrightarrow y_1 \wedge x_2 \Leftrightarrow y_2$.

[3] 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.

4 Temporal Logics

We start this section on temporal logics by a quick overview of CTL (Computation Tree Logic), the specification logic associated to symbolic model checking. We then define a hierarchy of proper sub-logics of CTL, named STL (Step Temporal Logic — Section 4.2), NF-CTL (No Fixpoint Temporal Logic — Section 4.3) and FF-CTL (Fewer Fixpoints Temporal Logic — Section 4.4). These logics are such that there is a proper inclusion of the corresponding sets of well-formed formulas.

The results in this paper apply to the verification of both NF-CTL and FF-CTL.

4.1 CTL: Computation Tree Logic

Syntax The set $T_{\mathsf{CTL}}(P)$ of Computation Tree Logic (CTL for short) formulas over a non-empty 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 $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{AX}f$, $\mathbf{EG}f$, $\mathbf{AG}f$, $\mathbf{EF}f$, $\mathbf{AF}f$, $\mathbf{E}[f\mathbf{U}g]$, $\mathbf{A}[f\mathbf{U}g]$, $\mathbf{E}[f\mathbf{R}g]$ and $\mathbf{A}[f\mathbf{R}g]$ are in $T_{\mathsf{CTL}}(P)$.

Each temporal logic operator is composed of:

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

Semantics The semantics of CTL is 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 $\forall i \geq 1, M, \pi(i) \models f$. 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 and $\exists i \geq 1, M, \pi(i) \models g \wedge \forall j, i > j \geq 1, M, \pi(j) \models f$. I.e. s is at the start of a path where g holds eventually and f holds until g becomes valid.

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

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

Two operators are of special interest in the context of this paper:

- **E**[**R**], where $M, s \models \mathbf{E}[f \mathbf{R} g]$ iff there exists a path π of M such that $\pi(1) = s$ and $\forall i \geq 1, M, \pi(i) \models f$ or $\exists i \geq 1, M, \pi(i) \models g \wedge \forall j, i \geq j \geq 1, M, \pi(j) \models f$. I.e. s is at the start of a path where f holds until g becomes valid (note that f must be valid in s and that g may never turn valid along π):

$$\mathbf{E}[g \mathbf{R} f] = \mathbf{E}[f \mathbf{U} (f \wedge g)] \vee \mathbf{EG} f;$$

- **A**[**R**], the universal counterpart of **E**[**R**].

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

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

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, as defined in [4]. For instance,

$$\mathbf{A}[f \mathbf{U} g] = \mathbf{lfp} Z [g \vee (f \wedge \mathbf{AX} Z)]$$

The number of iterations required to reach a fixpoint depends on two factors: the size of the model and the size of the formula.

4.2 STL: Step Temporal Logic

STL is a branching-time temporal logic that does not allow unbounded state quantifiers **G**, **F**, **U** and **R**. Therefore it only contains those formulas that assert exact timing properties, such as "in two steps, necessarily f", which can be written $\mathbf{AXAX}f$, or "in two to three steps, necessarily f" (expressed as $\mathbf{AXAX}(f \vee \mathbf{AX}f)$).

Formally, the set $T_{\mathsf{STL}}(P)$ of Step Temporal Logic (STL for short) formulas over a set of propositions P is the smallest set such that $P \subseteq T_{\mathsf{STL}}(P)$ and, if f and g are in $T_{\mathsf{STL}}(P)$, then $\neg f$, $f \wedge g$ and $\mathbf{EX}f$ are in $T_{\mathsf{STL}}(P)$[3]. Note that $T_{\mathsf{STL}}(P) \subset T_{\mathsf{CTL}}(P)$ and the semantics of STL formulas with respect to a Kripke structure is the same as corresponding CTL formulas.

The symbolic model checking algorithm for CTL also applies to STL. However, none of the STL formulas requires a fixpoint computation. The number of iterations is bounded by the size of the formula (more precisely, it is equal to the number of **EX** operators in the formula), and is independent of the size of the model.

4.3 NF-CTL: No Fixpoint Computation Tree Logic

NF-CTL is a branching time temporal logic whose terms can be model checked without any fixpoint computation. Below, we formally define NF-CTL and discuss its expressive power.

Definition The set $T_{\mathsf{NF\text{-}CTL}}(P)$ of Iterationless Computation Tree Logic (NF-CTL for short) formulas over a set of propositions P is the smallest set such that:

- $T_{\mathsf{STL}}(P) \subseteq T_{\mathsf{NF\text{-}CTL}}(P)$ and,
- if f and g are in $T_{\mathsf{STL}}(P)$, then $\mathbf{E}[g\mathbf{R}f]$, $\mathbf{A}[g\mathbf{R}f]$, $\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$ and $\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$ are in $T_{\mathsf{NF\text{-}CTL}}(P)$.
- if f and g are in $T_{\mathsf{NF\text{-}CTL}}(P)$, then $\neg f$, $f \wedge g$, are in $T_{\mathsf{NF\text{-}CTL}}(P)$.

Note that $T_{\mathsf{NF\text{-}CTL}}(P) \subset T_{\mathsf{CTL}}(P)$ and the semantics of NF-CTL formulas with respect to a Kripke structure is the same as corresponding CTL formulas.

In the following, we call $\mathbf{A}[g\mathbf{R}f]$ and $\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$ *universal NF-CTL* formulas; $\mathbf{E}[g\mathbf{R}f]$ and $\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$, *existential NF-CTL* formulas; and $\neg f$ and $f \wedge g$, *connective NF-CTL* formulas.

Expressive Power Although at first look, NF-CTL seems restricted, important classes of CTL formulas can be expressed in this subset:

- always: $\mathbf{AG}f = \mathbf{A}[\mathit{false}\mathbf{R}f]$;
- almost always: $\mathbf{AG}(f \Rightarrow \mathbf{AG}f) = \mathbf{AG}(f \Rightarrow \mathbf{A}[\mathit{false}\mathbf{R}f])$;

[3] Here and in the following definitions of temporal logics, we use the standard equations to relate basic operators **EX**, **EG** and **E**[**U**] with the other operators. The same remark applies to boolean operators.

- release: $\mathbf{A}[g\mathbf{R}f]$;
- handshaking: $\mathbf{AG}(Req \Rightarrow \mathbf{A}[Ack\mathbf{R}Req])$;
- eventually: $\mathbf{AF}f = \neg\mathbf{EG}\neg f = \neg\mathbf{E}[false\mathbf{R}f]$.

4.4 FF-CTL: Fewer Fixpoints Computation Tree Logic

FF-CTL is a branching time temporal logic whose terms can be model checked with fewer fixpoint computations than in McMillan's algorithm. FF-CTL basically consists of CTL formulas where some subterms not embedded in temporal operators, are NF-CTL formulas.

Definition The set $T_{\mathsf{FF\text{-}CTL}}(P)$ of Fewer Fixpoints Computation Tree Logic (FF-CTL for short) formulas over a set of propositions P is the smallest set such that:

- $T_{\mathsf{NF\text{-}CTL}}(P) \subseteq T_{\mathsf{FF\text{-}CTL}}(P)$ and,
- if f and g are in $T_{\mathsf{CTL}}(P)$, then $\mathbf{E}[g\mathbf{R}f]$, $\mathbf{A}[g\mathbf{R}f]$, $\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$ and $\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$ are in $T_{\mathsf{FF\text{-}CTL}}(P)$.
- if f is in $T_{\mathsf{FF\text{-}CTL}}(P)$ and g in $T_{\mathsf{CTL}}(P)$, then $\neg f$, $f \wedge g$, are in $T_{\mathsf{FF\text{-}CTL}}(P)$.

Note that $T_{\mathsf{NF\text{-}CTL}}(P) \subset T_{\mathsf{FF\text{-}CTL}}(P) \subset T_{\mathsf{CTL}}(P)$ and the semantics of FF-CTL formulas for a given Kripke structure is the same as the corresponding CTL formulas.

Expressive Power To evaluate the practical expressive power of FF-CTL, we have made a small statistical study based on a publicly available list of SMV examples [16]. On a total of 107 CTL formulas, 41 (38%) are syntactically NF-CTL formulas, and 49 (46%) are syntactically FF-CTL formulas, even though the $\mathbf{A}[\mathbf{R}]$ and $\mathbf{E}[\mathbf{R}]$ operators are not directly available in SMV specification language.

5 Sufficient Conditions for FF-CTL Formulas

Since FF-CTL is a subset of CTL, standard model checking algorithms for CTL formulas can also be used to check FF-CTL formulas. However, in this section we show heuristics for verifying FF-CTL formulas with fewer fixpoint computations. In the following, for each class of FF-CTL formulas, we present sufficient conditions that guarantee they are valid of a Kripke structure. We then provide proofs for these conditions and, in section 6, show how an alternative model checking algorithm can make use of these results.

5.1 Conditions

Table 1 presents sufficient conditions for universal FF-CTL formulas to be true of a Kripke structure. For instance, if $Forward(f \wedge \neg g) \subseteq f$ holds, then we can conclude that the FF-CTL formula $\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$ is valid. If, additionally,

$I \models f$ holds, then $\mathbf{A}[g\mathbf{R}f]$ is also valid. When $I \not\models f$, then $\mathbf{A}[g\mathbf{R}f]$ does not hold. Otherwise, we cannot conclude.

Standard model checking verification of these formulas requires one (in the case of $\mathbf{A}[g\mathbf{R}f]$) or two (in the case of $\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$) additional fixpoint computations than testing the above requirements. Particularly, if both f and g are STL formulas, no fixpoint computation is needed to test the sufficient conditions.

$I \subseteq f$	$Forward(f \wedge \neg g) \subseteq f$	$\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$	$\mathbf{A}[g\mathbf{R}f]$
false	*false*	?	*false*
false	*true*	*true*	*false*
true	*false*	?	?
true	*true*	*true*	*true*

Table 1. Sufficient conditions for universal FF-CTL formulas

Table 2 gives sufficient conditions for existential FF-CTL formulas to hold in a Kripke structure. Again, the interesting part of this table lays in the second and fourth lines. If it can be shown that $f \wedge \neg g \subseteq Backward(f)$ is true, we can conclude that the FF-CTL formula $\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$ is valid. Furthermore, if we have $I \models f$, then $\mathbf{E}[g\mathbf{R}f]$ is also valid. Again, standard model checking verification of these formulas requires more fixpoint computations than checking the above requirements. As above, if the formula also belongs to NF-CTL, then no fixpoint computation is needed at all.

$I \subseteq f$	$f \wedge \neg g \subseteq Backward(f)$	$\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$	$\mathbf{E}[g\mathbf{R}f]$
false	*false*	?	*false*
false	*true*	*true*	*false*
true	*false*	?	?
true	*true*	*true*	*true*

Table 2. Sufficient conditions for existential FF-CTL formulas

5.2 Proofs for Table 1

Proposition 1. *Let $M = (S, T, I, L)$ be a Kripke structure. If the image of $f \wedge \neg g$ in M is a subset of f ($Forward(f \wedge \neg g) \subseteq f$), then $M \models \mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$.*

Proof. We prove the contrapositive of this formula, that is:

> If a Kripke structure M is such that $M \models \neg\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$ then $Forward(f \wedge \neg g) \not\subseteq f$.

First, we rewrite formula $\neg\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$ as follows:

$$\begin{aligned}
\neg\mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f]) &\equiv \neg\neg\mathbf{EF}(\neg(f \Rightarrow \mathbf{A}[g\mathbf{R}f])) \\
&\equiv \mathbf{EF}(\neg(f \Rightarrow \mathbf{A}[g\mathbf{R}f])) \\
&\equiv \mathbf{EF}(\neg(\neg f \vee \mathbf{A}[g\mathbf{R}f])) \\
&\equiv \mathbf{EF}(f \wedge \neg\mathbf{A}[g\mathbf{R}f])) \\
&\equiv \mathbf{EF}(f \wedge \neg(\neg\mathbf{E}[\neg g\mathbf{U}\neg f])) \\
&\equiv \mathbf{EF}(f \wedge \neg(\neg\mathbf{E}[\neg g\mathbf{U}\neg f])) \\
&\equiv \mathbf{EF}(f \wedge \mathbf{E}[\neg g\mathbf{U}\neg f])
\end{aligned}$$

The semantics of operator **EF** states that if $\mathbf{EF}(f \wedge \mathbf{E}[\neg g\mathbf{U}\neg f])$ holds, then, there is a finite path $\pi_1, \pi_2, \ldots \pi_n, n \geq 1$ of M such that:

$$\pi_1 \in I \text{ and } \pi_n \models f \wedge \mathbf{E}[\neg g\mathbf{U}\neg f].$$

Therefore,

$$\pi_n \models f \text{ and } \pi_n \models \mathbf{E}[\neg g\mathbf{U}\neg f].$$

The semantics of $\mathbf{E}[\mathbf{U}]$ states that if $\mathbf{E}[\neg g\mathbf{U}\neg f]$ holds for π_n, then, there is a finite path $\pi_n, \pi_{n+1}, \ldots \pi_m, m \geq n$ of M such that:

$$\pi_m \models \neg f \text{ and } \forall i, n \leq i < m : \pi_i \models f \wedge \neg g.$$

Since $\pi_n \models f$, then $\pi_m \neq \pi_n$ and $m > n$. Therefore $\pi_{m-1} \models f \wedge \neg g$ and $\pi_m \models \neg f$. In conclusion, $Forward(f \wedge \neg g) \not\subseteq f$. □

Proposition 2. *Let $M = (S, T, I, L)$ be a Kripke structure. If the image of $f \wedge \neg g$ in M is a subset of f ($Forward(f \wedge \neg g) \subseteq f$), and f is valid for all initial states ($I \subseteq f$), then $M \models \mathbf{A}[g\mathbf{R}f]$.*

Proof. By hypothesis, in M, $Forward(f \wedge \neg g) \subseteq f$ holds and Proposition 1 applies. Therefore, $M \models \mathbf{AG}(f \Rightarrow \mathbf{A}[g\mathbf{R}f])$, and, by weakening, we can infere that $M \models f \Rightarrow \mathbf{A}[g\mathbf{R}f]$. Since, by hypothesis, $M \models f$, through *modus ponens* we get $M \models \mathbf{A}[g\mathbf{R}f]$. □

5.3 Proofs for Table 2

Lemma 1. *Let f and g be two arbitrary CTL formulas. Then,*

$$\mathbf{A}[f\mathbf{U}g] \equiv \mathbf{A}[(f \wedge \neg g)\mathbf{U}g]$$

Proof. Using the fixpoint characterization of operator $\mathbf{A}[\mathbf{U}]$, the proof of this lemma is straightforward:

$$\begin{aligned}
\mathbf{A}[f\mathbf{U}g] &\equiv \mathbf{lfp}K[g \vee (f \wedge \mathbf{AX}K)] \\
&\equiv \mathbf{lfp}K[g \vee (f \wedge \neg g \wedge \mathbf{AX}K)] \\
&\equiv \mathbf{A}[(f \wedge \neg g)\mathbf{U}g]
\end{aligned}$$

Proposition 3. *Let M be a Kripke structure. If the inverse image of f in M is a subset of $f \wedge \neg g$ ($f \wedge \neg g \subseteq Backward(f)$), then $M \models \mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$.*

Proof. Again, we prove the contrapositive of this formula, that is:

If a Kripke structure M is such that $M \models \neg\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$ then

$$f \wedge \neg g \not\subseteq Backward(f).$$

We shall show that whenever $M \models \neg\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$, there exists a state of M where $f \wedge \neg g$ holds and f holds in none of its successors.

First, we rewrite formula $\neg\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$ as follows:

$$\begin{aligned}
\neg\mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f]) &\equiv \mathbf{EF}\neg(f \Rightarrow \mathbf{E}[g\mathbf{R}f]) \\
&\equiv \mathbf{EF}(f \wedge \neg\mathbf{E}[g\mathbf{R}f]) \\
&\equiv \mathbf{EF}(f \wedge \mathbf{A}[\neg g\mathbf{U}\neg f]) \\
&\equiv \mathbf{EF}(f \wedge (\neg f \vee (\neg g \wedge \mathbf{AXA}[\neg g\mathbf{U}\neg f]))) \\
&\equiv \mathbf{EF}(f \wedge \neg g \wedge \mathbf{AXA}[\neg g\mathbf{U}\neg f])
\end{aligned}$$

The semantics of operator **EF** states that if $\mathbf{EF}(f \wedge \neg g \wedge \mathbf{AXA}[\neg g\mathbf{U}\neg f])$ holds, then, there is a finite path $\pi_1, \pi_2, \ldots \pi_n, n \geq 1$ of M such that:

$$\pi_1 \in I \text{ and } \pi_n \models f \wedge \neg g \wedge \mathbf{AXA}[\neg g\mathbf{U}\neg f].$$

Therefore, using Lemma 1:

$$\pi_n \models f \wedge \neg g \text{ and } \pi_n \models \mathbf{AXA}[(f \wedge \neg g)\mathbf{U}\neg f].$$

Let $\pi = \pi_n, \pi_{n+1}, \ldots$ be a path starting at state π_n and let $l(\pi) = \min(\{k | \pi_k \models \neg f\})$ be the index of the first state of such path where $\neg f$ holds. Note that, on any path π starting at π_n, the formula $\mathbf{A}[(f \wedge \neg g)\mathbf{U}\neg f]$ is valid in states $\pi_{n+1}, \ldots \pi_{l(\pi)}$.

Let $m = \max(\{l(\pi) \mid \pi = \pi_n, \pi_{n+1}, \ldots\})$. So $m - n$ is the length of the longest path prefix (c.f. fig. 3), starting at π_n, such that $f \wedge \neg g$ holds on every state $\pi_n, \pi_{n+1}, \ldots \pi_{m-1}$. $\neg f$ holds in state π_m.

Note that $\pi_{m-1} \models \mathbf{A}[(f \wedge \neg g)\mathbf{U}\neg f]$ and $\pi_{m-1} \models \neg f$. Therefore:

$$\pi_{m-1} \models \mathbf{AXA}[(f \wedge \neg g)\mathbf{U}\neg f]$$

Let π'_m be an arbitrary successor of π_{m-1}: $\pi'_m \models \mathbf{A}[(f \wedge \neg g)\mathbf{U}\neg f]$.

Suppose $\pi'_m \models f$, and consequently $\pi'_m \models f \wedge \neg g$. Therefore the path prefix $\pi_n\pi_{n+1}\ldots\pi_{m-1}\pi'_m$ starts at state π_n, is of length $1 + m - n$, and $f \wedge \neg g$ holds on every state of the path, which is a contradiction. In conclusion, no successor π'_m of π_{m-1} is such that $\pi'_m \models f$.

Since $\pi_{m-1} \models f \wedge \neg g$, then $f \wedge \neg g \not\subseteq Backward(f)$,

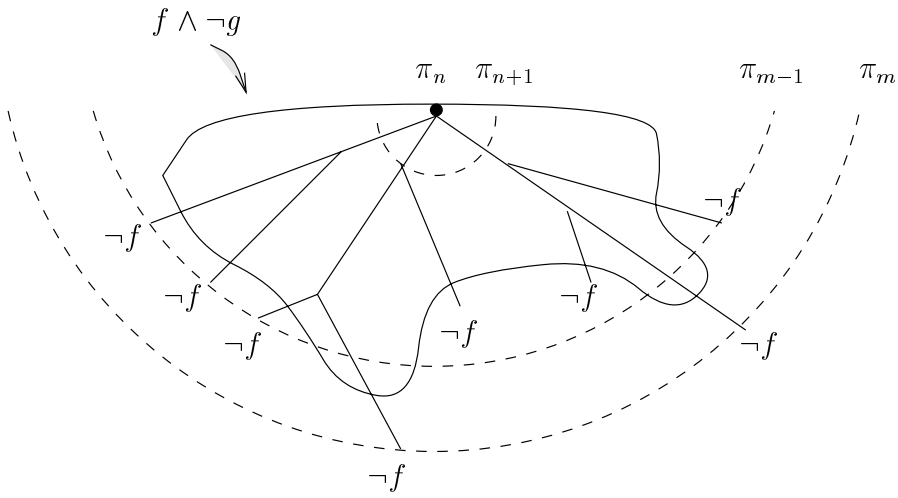

Fig. 3. Paths from π_n where $f \wedge \neg g$ holds

Proposition 4. *Let M be a Kripke structure. If the inverse image of f in M is a subset of $f \wedge \neg g$ ($f \wedge \neg g \subseteq Backward(f)$), and f is valid for all initial states ($I \subseteq f$), then $M \models \mathbf{E}[g\mathbf{R}f]$.*

The argument is similar to proof of Proposition 2:

Proof. By hypothesis, $f \wedge \neg g \subseteq Backward(f)$ holds in M. Therefore, Proposition 3 applies and $M \models \mathbf{AG}(f \Rightarrow \mathbf{E}[g\mathbf{R}f])$. By weakening, we can infere that $M \models f \Rightarrow \mathbf{E}[g\mathbf{R}f]$. Since $M \models f$ (by hypothesis), through *modus ponens* we get $M \models \mathbf{E}[g\mathbf{R}f]$.

6 Algorithms

Figure 4 contains the new CTL symbolic model checking algorithm, named *ModelCheck*. It takes as parameters m, a BDD-based representation of a finite-state transition system, and f, a CTL formula. $m.I$ denotes the characteristic function of the initial states of the system.

ModelCheck uses three subroutines:

- $ModelCheck_{\mathsf{FF\text{-}CTL}}$ is presented in Figure 5 and is further detailed in the following. It basically implements symbolic model checking for FF-CTL, and yields a boolean result.
- $ModelCheck_{\mathsf{CTL}}$ is McMillan's symbolic model checking algorithm. Given a BDD-based representation of the transition system and a CTL formula, it returns the BDD of the characteristic function of the set of states where the formula is valid.

– *bdd_implies* is a boolean predicate and tests if its first argument implies its second argument (both arguments are BDDs).

Line 3, $ModelCheck$ tests if there is a FF-CTL formula f' equivalent to f. If this is the case, then $ModelCheck_{\mathsf{FF\text{-}CTL}}$ is called with f' (line 4), otherwise $ModelCheck_{\mathsf{CTL}}$ is called with f, and the result is checked against the initial states of M (line 6).

```
1 function ModelCheck(m : Model, f : T_CTL) : boolean
2 begin
3     if ∃f' ∈ T_FF-CTL | f' ≡ f then
4         return ModelCheck_FF-CTL(m, f')
5     else
6         return bdd_implies(m.I, ModelCheck_CTL(m, f))
7 end
```

Fig. 4. Algorithm for $ModelCheck$

$ModelCheck_{\mathsf{FF\text{-}CTL}}$, the symbolic model checking algorithm for FF-CTL, is given in Figure 5. Its parameters are m, a BDD-based representation of a finite-state transition system, and f, a FF-CTL formula. f is matched against the different possible patterns of FF-CTL formulas, and action is taken accordingly:

– When f is a negation (lines 6 and 7), the result is the negation of the model checking for the formula argument f_1.
– When f is a conjunction (lines 8 and 9), the result is the conjunction of the model checking for the formula arguments f_1 and f_2.
– When f is a $\mathbf{A}[f_2\mathbf{R}f_1]$ universal FF-CTL formula (lines 10 to 19), we first check that f_1 is valid in the initial states. If not, the formula is already proven false in m, otherwise we go on with the verification by model checking f_2 and testing the corresponding sufficient condition. If this condition is verified, the result is *true*. Otherwise, we cannot conclude, and McMillan's symbolic model checking is invoked on f.
– If on the other hand, f is a $\mathbf{AG}(f_1 \Rightarrow \mathbf{A}[f_2\mathbf{R}f_1])$ universal FF-CTL formula (lines 20 to 26), both arguments f_1 and f_2 are model checked accordingly and the corresponding sufficient condition is then tested. If verified, the result is *true*. Otherwise, we cannot conclude, and McMillan's symbolic model checking is invoked on f[4].
– The procedure is much the same when f is an existential FF-CTL formula (lines 27 to 43).

[4] It is important to note that using a standard *caching* policy, the result of model checking subformulas of f is kept in a table and reused when needed in the verification of f.

```
1  function ModelCheck_FF-CTL(m : Model, f : T_FF-CTL) : boolean
2  var
3      f1, f2 : T_CTL
4      F1, F2 : BDD
5  begin
6      if f = ¬f1 then
7          return ¬ModelCheck(m, f1)
8      elsif f = f1 ∧ f2 then
9          return ModelCheck(m, f1) ∧ ModelCheck(m, f2)
10     elsif (f = A[f2 R f1]) then      --Table 1, Column 4
11         F1 ← ModelCheck_CTL(m, f1)
12         if bdd_implies(m.I, F1) then
13             F2 ← ModelCheck_CTL(m, f2)
14             if bdd_implies(m.I, Forward(m, bdd_and(F1, bdd_not(F2))), F1) then
15                 return true      --Table 1, Column 4, Line 4
16             else
17                 return ModelCheck_CTL(m, f)
18         else
19             return false
20     elsif f = AG(f1⇒A[f2 R f1]) then      --Table 1, Column 3
21         F1 ← ModelCheck_CTL(m, f1)
22         F2 ← ModelCheck_CTL(m, f2)
23         if bdd_implies(m.I, Forward(m, bdd_and(F1, bdd_not(F2))), F1) then
24             return true      --Table 1, Column 3, Line 4
25         else
26             return ModelCheck_CTL(m, f)
27     elsif (f = E[f2 R f1]) then      --Table 2, Column 4
28         F1 ← ModelCheck_CTL(m, f1)
29         if bdd_implies(m.I, F1) then
30             F2 ← ModelCheck_CTL(m, f2)
31             if bdd_implies(m.I, bdd_and(F1, bdd_not(F2)), Backward(m, F1)) then
32                 return true      --Table 2, Column 4, Line 4
33             else
34                 return ModelCheck_CTL(m, f)
35         else
36             return false
37     elsif f = AG(f1⇒E[f2 R f1]) then      --Table 2, Column 3
38         F1 ← ModelCheck_CTL(m, f1)
39         F2 ← ModelCheck_CTL(m, f2)
40             if bdd_implies(m.I, bdd_and(F1, bdd_not(F2)), Backward(m, F1)) then
41             return true      --Table 2, Column 3, Line 4
42         else
43             return ModelCheck_CTL(m, f)
44 end
```

Fig. 5. Algorithm for $ModelCheck_{\mathsf{FF\text{-}CTL}}$

6.1 Complexity

To validate our approach, we shall compare the complexity of our algorithm $ModelCheck$ with McMillan's $ModelCheck_{\mathsf{CTL}}$.

The best case occurs when f is a FF-CTL formula and the corresponding sufficient conditions hold. In this case, it saves a fixpoint computation of length at most linear in the size of M (i.e. $|S|$, the number of states of M). Since, at each step of the computation, $ModelCheck_{\mathsf{CTL}}$ invokes $Backward$ or $Forward$ once, the complexity of $ModelCheck$ is $|S|$ times better ($Forward$ and $Backward$ are equally complex).

The worst case occurs when f is a FF-CTL formula and the corresponding sufficient conditions do not hold. In this case, the additional cost is that of computing of the sufficient conditions, which is that of $Backward$. The resulting complexity of the whole algorithm is still linear in $|S|$. Moreover, it is possible to optimize the algorithm and reuse the backward computation required in testing the sufficient condition, as a first step in the fixpoint computation of $ModelCheck_{\mathsf{CTL}}$.

7 Conclusion and Future Work

We have defined NF-CTL, a subset of CTL formulas that can be checked by testing simple sufficient conditions, and do not involve any fixpoint computation, as opposed to previously published symbolic model checking for CTL [14] or subsets thereof [11].

We then identified FF-CTL, a second, larger, subset of CTL that can by verified using fewer fixpoint computations than [14]. Based on these results, we developed a new model checking algorithm for CTL that tests the identified sufficient conditions wherever relevant and uses [14] otherwise. The complexity of the new algorithm with respect to [14] is then discussed.

We have started to implement the proposed model checking algorithm into existing verification tools that use [14]. Preliminary results show that the incurred time penalty is so small that it cannot be quantified (that is, is smaller than the precision of the time measurement of our computing platforms). Also, in the case of invariant formulas (a special case of FF-CTL formulas), we often get positive results when testing sufficient conditions, and therefore are able to save fixpoint computations. We plan to complete this study considering the whole FF-CTL logic. We are also studying a rewrite system for CTL that finds equivalent formulas in FF-CTL wherever possible, and otherwise attempts to find an equivalent CTL formula with fewer temporal operators.

References

[1] 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.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS'99*, Lecture Notes in Computer Science. Springer Verlag.

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

[4] 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.

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

[6] E.M. Clarke, O. Grumberg, and D.Long. Model checking and abstraction. In *19th Annual ACM Symposium on Principles of Programming Languages*, 1992.

[7] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. In *L. Claesen, editor, 11th International Symposium on Computer Hardware Description Languages: CHDL'93*. NorthHolland, 1993.

[8] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking.

[9] D. Déharbe. *Vérification Symbolique de Modèle pour la logique CTL: Étude et Application au Langage VHDL*. PhD thesis, Université Joseph Fourier-Grenoble 1, Nov. 1996.

[10] D. Déharbe and A. Martins Moreira. Using induction and BDDs to model check invariants. In Hon F. Li and David K. Probst, editors, *CHARME'97: Correct Hardware Design and Verification Methods*, Montréal, Canada, Oct. 1997. Chapman & Hall.

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

[12] Bernhard Josko. MCTL – an extension of CTL for modular verification of concur rent systems. pages 165–187, 1987.

[13] J.C. Madre. *PRIAM Un outil de vérification formelle des circuits intégrés digitaux*. PhD thesis, Ecole nationale supérieure des télécommunications, Paris, France, June 1990. 90 E 007.

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

[15] 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.

[16] B. Yang. Fmcad'98 benchmark traces. `http://www.cs.cmu.edu/~bwolen/fmcad98`.