Evaluation of SAT-based Bounded Model Checking of ACTL Properties *

Yanyan Xu[1,2], Wei Chen[1,2], Liang Xu[1,2] and Wenhui Zhang[1]
[1]State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]School of Information Science and Engineering,
Graduate University of the Chinese Academy of Sciences, Beijing, China
xuyy@ios.ac.cn

Abstract

Bounded model checking (BMC) based on SAT has been introduced as a complementary method to BDD-based symbolic model checking of LTL and ACTL properties in recent years. For general LTL and ACTL properties, BMC has traditionally aimed mainly at error detection, taking the advantage that error detection may only need to explore a small portion of the whole state space. Recently bounded model checking aiming at verification has also been proposed. The aim of this paper is to exploit the strength of BMC methods by combining different BMC approaches and compare it with the traditional BDD-based symbolic methods. We consider two bounded model checking methods, which are for error detection and verification of ACTL properties, respectively, and then combine them to a BMC algorithm. Based on this algorithm, we have implemented a tool named BMV (bounded model verifier), and carried out a number of experiments, and we have then compared BMV with Cadence SMV. The experimental results show that for certain types of problems, both for verification and error detection, BMV can perform much better than Cadence SMV in both time and memory consumption, and we believe that this is the first attempt to have an implementation of a method that combines practical error detection and verification of ACTL properties by SAT-based model checking.

1 Introduction

A common method used in formal verification is model checking [7, 6]. Its basic idea consists of representing a program or a system as a Kripke structure, representing a specification as a temporal logic formula, and checking automatically whether the formula holds in the model [15]. Generally, Binary Decision Diagrams [1] are used to symbolically represent the transition relations and sets of states. This approach, known as symbolic model checking [2], has been successfully applied in practice. However, BDDs are very sensitive to the type and size of the system. Therefore much effort has been put into the research aiming at minimizing models. The methods include semantic minimization [17], abstraction techniques [11], partial order reductions [18], symmetry reductions [8], compositional techniques for splitting verification tasks [3], case-based partition techniques [13, 21], and so on.

Due to advances in algorithms and tools for the Boolean satisfiability problem (SAT) [14], formal reasoning based on SAT is proven to be an alternative to BDDs, and bounded model checking based on SAT has been introduced as a complementary method to BDD-based symbolic model checking of LTL and ACTL properties [4, 5, 16, 22, 23]. The basic idea of the approach presented in [4, 5, 16] is to search for a counterexample of a particular bound k. First, the approach generates a propositional formula, and then a SAT solver will be used to check the formula, and if the formula is satisfied by the SAT solver, it means a counterexample exists, so the approach is designed to find errors. For checking systems that are error free with respect to given properties, the approach is not practical because the bound k needed to be checked is usually too big (general over approximations of the bounds for LTL and ACTL are respectively $|M| \cdot 2^{|\varphi|}$ and $|M|$, where $|M|$ is the size of the model and $|\varphi|$ is the size of the formula). In order to solve the problem, [22, 23] proposes methods that can partly avoid the dependence on such a bound for verification.

The aim of our paper is to exploit the strength of SAT-based bounded model checking methods via comparison with the traditional BDD-based symbolic methods. We consider two bounded model checking methods [16, 23], which

*Supported by the National Natural Science Foundation of China under Grant No. 60573012 and 60421001, and the National Grand Fundamental Research 973 Program of China under Grant No. 2002cb312200.

are for error detection and verification of ACTL properties respectively, and then combine them to a BMC algorithm. We have implemented a tool named BMV (bounded model verifier) based on this algorithm, and carried out a number of experiments, and then we make a comparison of BMV and Cadence SMV [24]. Our experiments show that for certain types of problems, BMV can perform much better than Cadence SMV in both time and memory consumption for ACTL properties. The reason why we choose ACTL for this first attempt to have an implementation of a method that combines practical error detection and verification by SAT-based model checking is that the number of iterations in the bounded model checking needed for proving or disproving ACTL properties is usually smaller than that for similar LTL properties, and this number is important for the practical efficiency of the method.

The rest of our paper is organized as follows. In section 2, we introduce the computation tree logic. In section 3, we discuss the SAT-based bounded model checking algorithm. Then, in section 4, we describe our bounded model checking tool BMV, and we give the experimental results of BMV and Cadence SMV in section 5. We conclude this paper and discuss our future work in section 6.

2 Computation Tree Logic

Computation tree logic (CTL) is a propositional branching-time temporal logic introduced by [7] as a specification language for finite state systems.

2.1 Syntax of CTL

Let AP be the set of atomic proposition symbols containing $true$. The syntax of CTL formulas is given by the following rules:

- If $p \in AP$, then p is a CTL formula.
- If φ and ψ are CTL formulas, then $\neg\varphi$, $\varphi \vee \psi$ and $\varphi \wedge \psi$ are CTL formulas.
- If φ and ψ are CTL formulas, then $EX\varphi$, $EG\varphi$ and $E(\varphi U \psi)$ are CTL formulas.

Additional logical connectives and modal operators of CTL can be defined as follows:

- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- $EF\varphi \equiv E(TrueU\varphi)$
- $E(\varphi R\psi) \equiv E(\psi U(\varphi \wedge \psi)) \vee EG\psi$
- $AF\varphi \equiv \neg EG(\neg\varphi)$
- $A(\varphi R\psi) \equiv \neg E(\neg\varphi U\neg\psi)$
- $AX\varphi \equiv \neg EX(\neg\varphi)$
- $AG\varphi \equiv \neg EF(\neg\varphi)$
- $A(\varphi U\psi) \equiv \neg E(\neg\varphi R\neg\psi)$

2.2 Semantics of CTL

A model for CTL formulas is a Kripke structure $\langle S, T, I, L\rangle$, where S is the set of states; $T \subseteq S \times S$ is the transition relation which is total; $I \subseteq S$ is a set of initial states; and $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state. A sequence $\pi = \pi_0\pi_1...$ of S is a path of M, if for every $i \geq 0$, $T(\pi_i, \pi_{i+1})$ holds.

Definition 2.1 *Let M be a model, s a state, p a proposition symbol, φ and ψ CTL formulas. $M, s \models \varphi$ denotes that φ is true at the state s in M. Let π be a path of M. The relation $\models$ is defined as follows:*

$$
\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s) \\
M, s \models \neg\varphi & \Leftrightarrow & M, s \not\models \varphi \\
M, s \models \varphi \wedge \psi & \Leftrightarrow & M, s \models \varphi \text{ and } M, s \models \psi \\
M, s \models \varphi \vee \psi & \Leftrightarrow & M, s \models \varphi \text{ or } M, s \models \psi \\
M, s \models EX\varphi & \Leftrightarrow & \exists\pi(\pi_0 = s \wedge M, \pi_1 \models \varphi) \\
M, s \models EG\varphi & \Leftrightarrow & \exists\pi(\pi_0 = s \wedge \forall k \geq 0 \\
 & & (M, \pi_k \models \varphi)) \\
M, s \models E(\varphi U\psi) & \Leftrightarrow & \exists\pi(\pi_0 = s \wedge \exists k \geq 0 \\
 & & (M, \pi_k \models \psi \wedge \forall j < k \\
 & & (M, \pi_j \models \varphi)))
\end{array}
$$

The restriction of CTL to E path quantifier such that implication is not used and negation is applied only to propositions is called ECTL, and the restriction of CTL to A path quantifier with the same restriction is called ACTL, and ACTL = $\{\neg\varphi \mid \varphi \in \text{ECTL}\}$.

Definition 2.2 *An ECTL formula φ is valid in M, denoted $M \models \varphi$, iff φ is true at some initial state of the model M.*

Definition 2.3 *An ACTL formula φ is valid in M, also denoted $M \models \varphi$, iff φ is true at every initial state of the model M.*

This definition expresses that in order to check whether an ACTL formula φ holds in M, we need to check whether φ holds for each of the initial states of M.

3 Bounded Model Checking Based on SAT

In this section, we introduce two sat-based bounded model checking methods for ACTL properties which are for error detection [16] and verification of valid properties [23] respectively. Both algorithms are about reducing the model checking problem ($M \models \varphi$) to the bounded model checking problem ($M_k \models_k \varphi$), and how $M_k \models_k \varphi$ relates to the satisfiability problem of the final encoding we get.

3.1 Bounded Semantics of ECTL

Let $M = \langle S, T, I, L\rangle$ be a model and $k \geq 0$. A k-path of M is a path $\pi = s_0...s_k$ where $(s_i, s_{i+1}) \in T$ for $i = 0, ..., k-1$. The k-model for M is a structure $M_k = \langle S, P_k, I, L\rangle$ where P_k is the set of all different k-paths of M. Let $\pi = \pi_0...\pi_k \in P_k$ and $loop(\pi)$ denotes $\bigvee_{i=0}^{k} T(\pi_k, \pi_i)$.

Definition 3.1 *Let M_k be a k-model, s a state, p a proposition symbol, and φ and ψ ECTL formulas. The relation $\models_k$ is defined as follows:*

$$
\begin{array}{lcl}
M_k, s \models_k p & \Leftrightarrow & p \in L(s) \\
M_k, s \models_k \neg p & \Leftrightarrow & p \notin L(s) \\
M_k, s \models_k \varphi \wedge \psi & \Leftrightarrow & M_k, s \models_k \varphi \text{ and } \\
 & & M_k, s \models_k \psi \\
M_k, s \models_k \varphi \vee \psi & \Leftrightarrow & M_k, s \models_k \varphi \text{ or } \\
 & & M_k, s \models_k \psi \\
M_k, s \models_k EX\varphi & \Leftrightarrow & \exists\pi(\pi_0 = s \wedge M_k, \\
 & & \pi_1 \models_k \varphi) \\
M_k, s \models_k EG\varphi & \Leftrightarrow & \exists\pi(\pi_0 = s \wedge loop(\pi) \\
 & & \wedge\forall 0 \leq i \leq k \\
 & & (M_k, \pi_i \models_k \varphi)) \\
M_k, s \models_k E(\varphi U\psi) & \Leftrightarrow & \exists\pi(\pi_0 = s \wedge \exists 0 \leq i \leq k \\
 & & (M_k, \pi_i \models_k \psi \wedge \forall j < i \\
 & & (M_k, \pi_j \models_k \varphi)))
\end{array}
$$

Since $EF\varphi \equiv E(trueU\varphi)$ and $E(\varphi R\psi) \equiv E(\psi U(\varphi \wedge \psi)) \vee EG\psi$, we only consider formulas of the form $\varphi \vee \psi$, $\varphi \wedge \psi$, $EX\varphi$, $EG\varphi$, $E(\varphi U\psi)$ constructed from propositions and the negation of propositions.

Definition 3.2 *An ECTL formula φ is valid in a k-model M_k, denoted $M_k \models_k \varphi$, iff φ is true at some initial state of the model M_k.*

Similarly, this definition is just definition 2.2 under bounded semantics.

3.2 Encoding the Model in SAT-Formulas

Let s be a vector of state variables, but in this paper, sometimes s is given some assignment, hence it means a state, and we can know this from its context. Let $k \geq 0$, N_k the number of different k-paths of M, and $s_{i,0}, ..., s_{i,k}$ mean a finite sequence of states on some path for each $i \in \{1, ..., N_k\}$. We show how to encode the model M_k into a propositional formula as below:

Definition 3.3 ***(Encoding the Model)*** *Let $M_k = \langle S, P_k, I, L\rangle$ be the k-model of M and φ an ECTL formula. The propositional formula $[M^{\varphi,s}]_k$ is defined as follows:*

$$[M^{\varphi,s}]_k := I(s) \wedge \bigwedge_{i=1}^{N_k} \bigwedge_{j=0}^{k-1} T(s_{i,j}, s_{i,j+1})$$

where I(s) represents a predicate that is only true when s is the initial state; and $T(s_{i,j}, s_{i,j+1})$ iff $(s_{i,j}, s_{i,j+1}) \in T$.

Now we have the encoding for the model M_k, and in next subsections we will have the encoding for properties.

3.3 Encoding Formulas for Error Detection

Let $p \in AP$ be a proposition symbol and $p(s)$ represent the propositional formula representing the states in which p is true according to L.

Definition 3.4 ***(Encoding Formulas for Error Detection)*** *Given a state s and a formula φ, the encoding $[\varphi, s]_k^e$ (e means for error detection) is defined as follows:*

$$
\begin{array}{lcl}
[p, s]_k^e & = & p(s) \\
[\neg p, s]_k^e & = & \neg p(s) \\
[\varphi \vee \psi, s]_k^e & = & [\varphi, s]_k^e \vee [\psi, s]_k^e \\
[\varphi \wedge \psi, s]_k^e & = & [\varphi, s]_k^e \wedge [\psi, s]_k^e \\
[EX\varphi, s]_k^e & = & \bigvee_{i=1}^{N_k}(s = s_{i,0} \wedge [\varphi, s_{i,1}]_k^e) \\
[EG\varphi, s]_k^e & = & \bigvee_{i=1}^{N_k}(s = s_{i,0} \wedge \bigvee_{l=0}^{k} L_{k,i}(l) \\
 & & \wedge \bigwedge_{j=0}^{k}[\varphi, s_{i,j}]_k^e) \\
[E(\varphi U\psi), s]_k^e & = & \bigvee_{i=1}^{N_k}(s = s_{i,0} \wedge \bigvee_{j=0}^{k}[\psi, s_{i,j}]_k^e \\
 & & \wedge \bigwedge_{t=0}^{j-1}[\varphi, s_{i,t}]_k^e)
\end{array}
$$

where $L_{k,j}(l) = T(s_{k,j}, s_{l,j})$ denotes a backward loop from the k-th state to the l-th state in the symbolic k-path j, for $0 \leq l \leq k$.

The number of different k-paths N_k considered in the encoding of the the model and the ECTL formulas is not a practical one for error detection. For practical application, the sufficient number of paths involved depends on k and the formula φ to be checked. Let $FORM$ be the set of ECTL formulas. [16] defined a function $f_k^e : FORM \rightarrow \mathbf{N}$ which can be used instead of N_k .

Definition 3.5 ***(f_k^e for Error Detection)*** *Define $f_k^e : FORM \rightarrow N$ for error detection as follows:*

- $f_k^e(p) = f_k^e(\neg p) = 0$, *if $p \in AP$;*
- $f_k^e(\varphi \wedge \psi) = f_k^e(\varphi) + f_k^e(\psi)$;
- $f_k^e(\varphi \vee \psi) = max(f_k^e(\varphi), f_k^e(\psi))$;
- $f_k^e(EX\varphi) = f_k^e(\varphi) + 1$;
- $f_k^e(EG\varphi) = (k+1) \cdot f_k^e(\varphi) + 1$;
- $f_k^e(E(\varphi U\psi)) = k \cdot f_k^e(\varphi) + f_k^e(\psi) + 1$.

Definition 3.6 *$[M^{\varphi,s}]_k^e$ is defined as Definition 3.3 except that $f_k^e(\varphi)$ is used instead of N_k.*

The encoding of the model checking problem is a combination of the encoding of the model and the formula.

Definition 3.7 $[M, \varphi, s]_k^e$:=$[M^{\varphi,s}]_k^e \wedge [\varphi, s]_k^e$

Now we will explain why it is correct that we translate the model checking problem $M \models \psi$ to the satisfiability checking problem of $[M^{\varphi,s}]_k^e \wedge [\varphi, s]_k^e$ step by step ($\varphi = \neg\psi$). Let M be a model, s be a state of M, M_k be a k-model of M, and φ be an ECTL formula, then [16] has proved:

- $M_k, s \models \varphi$ implies $M_l, s \models \varphi$, for $l \geq k$;
- $M_k, s \models \varphi$ implies $M, s \models \varphi$;
- $M \models \varphi \Leftrightarrow M_k \models_k \varphi$;

With these three points, we can get that:

- $M_k \models_k \varphi$ iff $[M, \varphi, s]_k^e$ is satisfiable;
- $M_k \models_k \neg\varphi$ iff $[M, \varphi, s]_k^e$ is unsatisfiable, for k=$|M|$.

With all these points, now we are able to translate the model checking problem to the bounded model checking problem and we can solve the latter problem with a state-of-the-art SAT solver. Please note the power of this method in finding errors largely depends on the fact that if $M \models \varphi$, there usually exists some $k < |M|$ such that $M_k \models_k \varphi$. Again, another fact that we have to reach the bound $k = |M|$ in order to check true ACTL formulas make it impossible to use this method for verification. However, with the encoding given in the next subsection, this may not be a problem.

3.4 Encoding Formulas for Verification

For the purpose of verification of valid ACTL properties, the encoding of ECTL formulas (the negation of the property to be verified) is a little different from that for error detection, and they are defined as follows.

Definition 3.8 ***(Encoding Formulas for Verification)*** *Given a state s and a formula φ. The encoding $[\varphi, s]_k^v$ (v means for verification) is defined as follows:*

$$
\begin{array}{lcl}
[p, s]_k^v & = & p(s) \\
[\neg p, s]_k^v & = & \neg p(s) \\
[\varphi \vee \psi, s]_k^v & = & [\varphi, s]_k^v \vee [\psi, s]_k^v \\
[\varphi \wedge \psi, s]_k^v & = & [\varphi, s]_k^v \wedge [\psi, s]_k^v \\
[EX\varphi, s]_k^v & = & \bigvee_{i=1}^{N_k} (s = s_{i,0} \wedge [\varphi, s_{i,1}]_k^v) \\
[EG\varphi, s]_k^v & = & \bigvee_{i=1}^{N_k} (s = s_{i,0} \wedge \bigwedge_{j=0}^{k} [\varphi, s_{i,j}]_k^v) \\
[E(\varphi U \psi), s]_k^v & = & \bigvee_{i=1}^{N_k} (s = s_{i,0} \wedge (\bigvee_{j=0}^{k} \\
 & & ([\psi, s_{i,j}]_k^v \wedge \bigwedge_{t=0}^{j-1} [\varphi, s_{i,t}]_k^v) \\
 & & \vee \bigwedge_{t=0}^{k} [\varphi, s_{i,t}]_k^v))
\end{array}
$$

where $[\varphi, s_{i,j}]_k$ denotes true, if $j > k$.

Similarly a calculated number depending on k and the formula to be verified can be used instead of N_k for the verification purpose.

Definition 3.9 ***(f_k^v for Verification)*** *Define $f_k^v : FORM \rightarrow N$ for verification as follows:*

- $f_k^v(p) = f_k^v(\neg p) = 0$, *if* $p \in AP$;
- $f_k^v(\varphi \wedge \psi) = f_k^v(\varphi) + f_k^v(\psi)$;
- $f_k^v(\varphi \vee \psi) = max(f_k^v(\varphi), f_k^v(\psi))$;
- $f_k^v(EX\varphi) = f_k^v(\varphi) + 1$;
- $f_k^v(EG\varphi) = (k+1) \cdot f_k^v(\varphi) + 1$;
- $f_k^v(E(\varphi U \psi)) = k \cdot f_k^v(\varphi) + max(f_k^v(\varphi), f_k^v(\psi)) + 1$.

Definition 3.10 *$[M^{\varphi,s}]_k^v$ is defined as Definition 3.3 except that $f_k^v(\varphi)$ is used instead of N_k.*

Definition 3.11 $[M, \varphi, s]_k^v$:=$[M^{\varphi,s}]_k^v \wedge [\varphi, s]_k^v$

Now we will explain how we can verify the true ACTL properties using the bounded model checking method. Similarly, let M be a model, s be a state of M, M_k be a k-model of M, and φ be an ECTL formula, then from [23], we know that:

- If $M_k \models_k \varphi$, then $[M, \varphi, s]_k^v$ is satisfiable, and this is equal to that if for some k, $[M, \varphi, s]_k^v$ is unsatisfiable, then $M_k \models_k \neg\varphi$.
- If $[M, \varphi, s]_{k+1}^v$ is satisfiable, then $[M, \varphi, s]_k^v$ is satisfiable; and this expresses the key idea of this encoding: for some k, if the encoding $[M, \varphi, s]_k^v$ is unsatisfiable, then $[M, \varphi, s]_l^v$ is unsatisfiable, for $l \geq k$. This is why this method is only useful for some of the operators. For some other operators like EF, it is not possible to get a useful encoding [23].

From the two points above, we can get:

- If $M \models \varphi$, then $[M, \varphi, s]_k^v$ is satisfiable for all $k \geq 0$.
- If for some k, $[M, \varphi, s]_k^v$ is unsatisfiable, then $M \models \neg\varphi$.

The second point is essential in the algorithm. We check the encoding $[M, \varphi, s]_k^v$, and if for some k, it is unsatisfiable, then we can tell that the ECTL property φ is false in M, so the ACTL one (ψ) is true.

3.5 The Bounded Model Checking Algorithm

Let M be a given model and ψ an ACTL formula, by combining the two methods described in subsection 3.3 and 3.4 together, we get a total bounded model checking algorithm and its pseudo code is displayed below.

01 $\quad$ *Let $\varphi = \neg\psi$, so φ is an ECTL formula*;

02 $for(k = 1; k \leq |M|; k++)\{$

03 $Calculate\ f_k^e(\varphi)$;

04 $Calculate\ f_k^v(\varphi)$;

05 $Encode\ [M^{\varphi,s_{0,0}}]_k^e$;

06 $Encode\ [M^{\varphi,s_{0,0}}]_k^v$;

07 $Encode\ [\varphi, s_{0,0}]_k^e$;

08 $Encode\ [\varphi, s_{0,0}]_k^v$;

09 $Check\ the\ satisfiability\ of\ [M, \varphi, s_{0,0}]_k^v$;

10 $If\ [M, \varphi, s_{0,0}]_k^v\ is\ unsatisfiable,\ report\ that\ M \models \psi\ is\ valid$;

11 $Check\ the\ satisfiability\ of\ [M, \varphi, s_{0,0}]_k^e$;

12 $If\ [M, \varphi, s_{0,0}]_k^e\ is\ satisfiable,\ report\ that\ M \models \psi\ does\ not\ hold$;

13 $If\ k = |M|\ is\ reached,\ report\ that\ M \models \psi\ is\ valid$;

14 }

Note that before we check the satisfiability of $[M, \varphi, s_{0,0}]_k^e$ and $[M, \varphi, s_{0,0}]_k^v$, we need to convert them to DIMACS [9] format, and this is done by defining one new variable for each OR-subtree.

The BMC algorithm can verify some valid properties before the completeness threshold $|M|$, while in the same time it can do error detection as before [16].

4 The Bounded Model Checking Verifier

The bounded model checking verifier (BMV) is a tool which could do either SAT-based bounded model checking for verification of ACTL[16, 23] and $\forall\mu$-calculus properties[19] or the traditional BDD-based model checking of μ-calculus properties[6]. In this paper, we only consider the BMC algorithm introduced in section 3. The input language of BMV is a first order transition system which allows the description of finite state systems that range from the detailed to the abstract. The language provides for modular hierarchical descriptions, and for the definition of reusable components. The only data types in the language are finite ones - Booleans, scalars and fixed arrays. After inputting a first order transition system and an ACTL formula, BMV can use the BMC algorithm to determine whether specification expressed in an ACTL formula is satisfied or not.

Let us consider the following program in the input language of BMV.

```
enum{ req, re, gr } etype;
enum{ 0..255 } byte;

MODULE MAIN
VAR
   bool x,y,t;
   etype c0[2];
   etype c1[2];
   byte m,n;
PROC
   P (c0,m);
   P (c1,n);
INIT
   x==false && y==false && t==false
   && m==0 && n==0
TRAN
   m==2 && c0[0]==req && y==true ->
   (x,y,m,c0[0]):=(true,true,m-1,c0[1]);
   m>=n && t==false ->
   (c1[0],t,x):=(gr,true,false);
END

MODULE P(etype c[2],byte l)
VAR
   byte lab;
INIT
   lab==0 && l==0
TRAN
   lab==0 && l==1 ->
   (c[1],l,lab):=(req,2,1);
   lab==1 && l<=2 ->
   (c[0],l,lab):=(re,l+1,2);
END
```

The model is a first order transition system, whose state is defined by a collection of state variables, which may be of Boolean or enumeration type. In the above program, for example, the variables x, y and t are declared to be Boolean type, while the variable type "etype" and "byte" are enumeration type which are introduced by the keyword **enum**. The value of an enumeration type variable is encoded by the interpreter using a collection of Boolean variables, so that the transition relation may be represented by propositional formulas. This encoding is invisible to the user, however. The input language of BMV also supports the array type whose base type can be either Boolean or enumeration type.

The initial state(s) of the transition system introduced by the keyword INIT is determined by conjunction of condi-

tions. In the above program, for example, in the MAIN process the initial state is set to be the state which satisfies the condition:
$x == false \;\&\&\; y == false \;\&\&\; t == false$
$\&\& \; m == 0 \;\&\&\; n == 0$.

The transition relation introduced by the keyword TRAN is determined by a collection of transition rules. A transition rule includes a condition statement and an assignment statement. For example:

$$m \geq n \;\&\&\; t == false \rightarrow (c1[0], t, x) := (gr, true, false);$$

The assignment statement on the right hand of '$\rightarrow$' will be executed if the condition statement on the left hand of '$\rightarrow$' is satisfied.

If a variable is not assigned in a program, the BMV system will be free to choose any legal value for the variable, giving it the characteristics of an unconstrained input to the system.

The above program also illustrates the definition of reusable modules. Notice that the module name "MAIN" has special meaning in BMV, in the same way that it does in the C++ programming language. A process can be instantiated several times with different arguments and these processes are introduced by the keyword PROC in the MAIN process.

Any of the members in the language including VAR, PROC, INIT and TRAN is optional, that is, they are not necessary and we use them just according to our requirement.

To summarize, the input language of BMV is designed to be flexible in terms of models it can describe. The language is designed to exploit the capabilities of model checking techniques. As a result, it has been made to support a particular model of communication between concurrent processes.

5 Experimental Results

We have carried out our experiments on three examples: the eight puzzle, a barrel shifter and a multiplier, and performed our experiments on the PC equipped with the processor P4 2.93GHz, 512MB main memory and the operating system Linux 2.6.12. We make use of the satisfiability solver MiniSat2 release 12-08-06 [25] , which uses the DIMACS format, in BMV. The test results of BMV are compared against results obtained from Cadence SMV [24] release 10-11-02, and we use the benchmark tool Run [26] to collect the memory consumption of Cadence SMV.

5.1 The Eight Puzzle (EP)

Our first example is the Eight Puzzle, the 3×3 version of the well-known sliding-tile puzzles (see Figure 1). There are eight numbered square tiles, and one empty position, called the "blank". Any tile horizontally or vertically adjacent to the blank can be swapped with the blank. The goal is to rearrange the tiles from some random initial configuration into a particular goal configuration. The initial configuration of our example is shown in the right graph of Figure 1, and in the graph, 0 means the "blank" position and 1-8 mean the eight square tiles. We number the nine positions p0-p8, which are shown in the left graph of Figure 1.

p0	p1	p2
p3	p4	p5
p6	p7	p8

0	1	2
3	4	5
6	7	8

Figure 1. The Eight Puzzle (EP)

We have tested two properties which are expressed by ACTL formulas. One is false and the other is true. The true formula is ψ_t=AXAF (n0=0 || n2=0 || n4=0 || n6=0), which means after a step, in the future, for all paths one of the four positions n0, n2, n4 and n6 will be the "blank" position. The false formula is ψ_f=AG $\neg$(n0=8 && n1=7 && n2=6 && n3=5 && n4=4 &&n5=3&& n6=2 && n7=1 && n8=0), which means for all paths, the problem will never reach the configuration illustrated in Figure 2.

8	7	6
5	4	3
2	1	0

Figure 2. The Final Configuration of EP

The test results of this example are shown in Table 1 (Cadence SMV) and Table 2 (BMV) respectively. For Cadence SMV, we present its time consumption (Time), memory consumption (Memory) and BDD nodes allocated (BDDs); and for BMV, we present the time consumption (Time), the memory consumption (Memory)[1], the bound k it needs (k), the number of variables (Vars) and the number of clauses

[1]The time and memory consumption only mean that actually used by MiniSat2. Overhead in converting the encoding into DIMACS format is not counted, because what really matters in BMC should be the time and memory consumption used by the SAT solver. Also, we only give the time

(Clas). In the tables below, the time unit is second and the memory unit is MB.

Formula	Time	Memory	BDDs
ψ_t	12.44	39.4	1609020
ψ_f	10.51	31.1	1098233

Table 1. Test Results of SMV for EP

Formula	Time	Memory	Vars	Clas	k
ψ_t	0.06	6.84	10563	29550	1
ψ_f	324.29	117.86	156220	436276	30

Table 2. Test Results of BMV for EP

From the experimental results of this example, we can see that for the true formula ψ_t, BMV performs much better than SMV in both time and memory consumption. However, for the false formula ψ_f, SMV outperforms BMV because the bound k BMV needs is too big (30).

5.2 A Barrel Shifter (BS)

The barrel shifter, which comes from [27], rotates the contents of a register file $B = \{b0, b1, b2, b3, b4\}$ with one position in each step. Let x' mean $next(x)$ for a variable x. The specification of BS is then as follows: $b0' = b4$, $b1' = b0$, $b2' = b1$, $b3' = b2$ and $b4' = b3$. The system also contains a fixed register file $R = \{r0, r1, r2, r3, r4\}$, and $r'0 = r0$, $r1' = r1$, $r2' = r2$, $r3' = r3$ and $r4' = r4$. At the initial state, we have $b0 = r0$, $b1 = r1$, $b2 = r2$, $b3 = r3$ and $b4 = r4$.

We have tested two properties which are expressed by ACTL formulas. One is false and the other is true. The true formula is ψ_t=AF ($b0 = r4 \rightarrow b1 = r0$), which means in the future, for all paths if $b0 = r4$, then $b1 = r0$. The false formula is ψ_f=AF $\neg(b0 = r4)$, which means in the future, for all paths, $b0 \neq r4$. The reason of that ψ_f is false is if $r0 = r1 = r2 = r3 = r4$, then for all paths, $b0 = r4$ is always true.

The test results of this example are shown in Table 3 (Cadence SMV) and Table 4 (BMV) respectively. "Type" in the tables means the type of the variables($b0 - b4$, $r0 - r4$). For example, 7 means that the variables can be assigned a value of 0-7. "-" means out of memory and the process is killed by Run [26]. The bound k BMV needs for ψ_t is 4 and 1 for ψ_f.

and memory consumption for the reported maximum bound k, so the time and memory consumption do not include the runs before k, i.e. from 1 to $k - 1$.

Formula	Type	Time	Memory	BDDs
ψ_t	7	8.23	47.8	1639097
ψ_t	8	19.54	131.1	5645213
ψ_t	9	33.45	218.8	8952805
ψ_t	10	57.23	300.9	13893024
ψ_t	11	88.94	448.4	18909523
ψ_t	12	-	-	-
ψ_f	7	8.53	48.6	1668700
ψ_f	8	25.43	134.6	5843618
ψ_f	9	37.84	223.1	9203016
ψ_f	10	59.14	308.5	14280617
ψ_f	11	91.47	453.3	19100620
ψ_f	12	-	-	-

Table 3. Test Results of SMV for BS

Formula	Type	Time	Memory	Vars	Clas	k
ψ_t	7	0.01	3.89	764	1988	4
ψ_t	8	0.01	4.06	942	2843	4
ψ_t	9	0.01	4.03	1000	3104	4
ψ_t	10	0.01	4.04	1058	3365	4
ψ_t	11	0.02	4.12	1116	3626	4
ψ_t	12	0.02	4.14	1174	3887	4
ψ_f	7	0.01	3.77	492	1478	1
ψ_f	8	0.01	3.90	588	2113	1
ψ_f	9	0.01	3.91	634	2320	1
ψ_f	10	0.01	3.90	680	2527	1
ψ_f	11	0.01	3.91	726	2734	1
ψ_f	12	0.01	4.05	772	2941	1

Table 4. Test Results of BMV for BS

From the experimental results of this example, we can see that both for the true formula ψ_t and for the false formula ψ_f, BMV performs much better than SMV in both time and memory consumption, and this is often the case when we have a small bound k in bounded model checking.

5.3 A Multiplier

The example, which also comes from [27], models an n×n bit shift-and-add multiplier. There are two arrays $f[n]$ and $i[n]$ in the models, and the models shifts the contents of $f[n]$ from left to right bit by bit while keeping $i[n]$ unchangeable.

We have tested the same property expressed by the ACTL formula ψ=AF$done$ according to two different multiplier models, and one is correct (the highest bit is set to 0), denoted by M_t, and the other is incorrect (the highest bit is set to 1), denoted by M_f.

The test results are shown in Table 5 (Cadence SMV)

and Table 6 (BMV) respectively. "Bit" in the tables means the number of the bits of the multiplier. For example, 16 means the multiplier is a 16×16 bit one, and k means the bound we need.

Model	Bit	Time	Memory	BDDs
M_t	16	10.83	69.0	2895362
M_t	17	23.4	133.6	5798471
M_t	18	50.46	263.7	11608541
M_t	19	-	-	-
M_f	16	11.44	69.0	2894677
M_f	17	24.43	133.8	5797688
M_f	18	53.07	264.1	11607575
M_f	19	-	-	-

Table 5. Test Results of SMV for the Multiplier

Model	Bit	Time	Memory	Vars	Clas	k
M_t	16	0.00	3.77	2913	4944	16
M_t	17	0.01	3.77	3282	5629	17
M_t	18	0.01	3.97	3673	6302	18
M_t	19	0.01	4.09	4086	7013	19
M_f	16	0.00	3.77	535	1006	1
M_f	17	0.00	3.77	568	1068	1
M_f	18	0.00	3.78	601	1130	1
M_f	19	0.00	3.77	634	1192	1

Table 6. Test Results of BMV for the Multiplier

From the experimental results, we can see that both for error detection and for verification, BMV performs much better than SMV in both time and memory consumption.

5.4 Experimental Evaluation

We have carried out our experiments on three examples: the eight puzzle, a barrel shifter and a multiplier. In Figures 3 and 4, there are eight graphs which depict the experimental results from Table 3, 4, 5 and 6. Figure 3 shows the results of the barrel shifter. The two top graphs show the time and memory consumption for ψ_t of SMV and BMV, and the graphs below show the same information for ψ_f. Figure 4 shows the results of the multiplier. The two top graphs show the time and memory consumption for M_t of SMV and BMV, and the graphs below show the same information for M_f.

These graphs illustrate that for these two problems, and for both error detection and verification, BMV performs much better than SMV in both time and memory consumption (the graphs of the resource consumption of BMV are quite close to the horizontal axis and almost invisible in the figures).

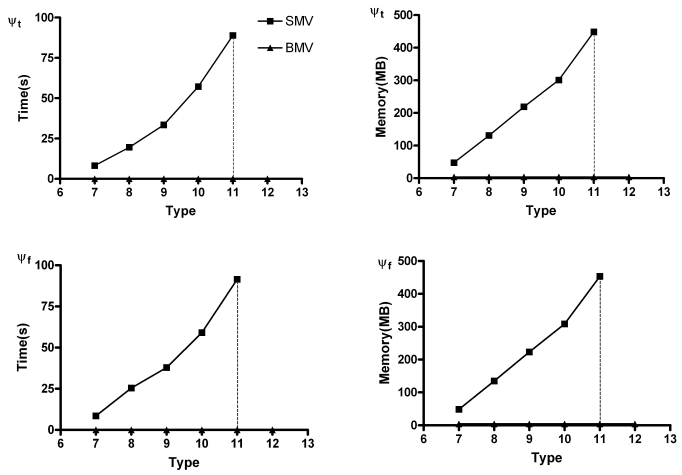

Figure 3. Graphs for BS

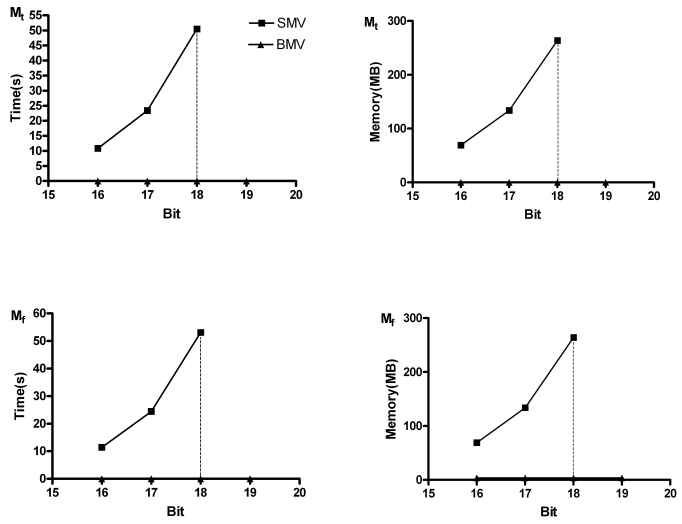

Figure 4. Graphs for Multiplier

6 Conclusions and Future Work

The basic idea of Bounded model checking is similar to that for searching finite models [20], and in the bounded model checking approach, we search for counter models of given sizes until we find one, and so it is designed to

find errors. However, in this paper, we have considered two bounded model checking methods [16, 23], which are for finding errors and verification respectively, and combined them to a BMC algorithm. Then we have implemented a tool named BMV (bounded model verifier) based on this algorithm, and carried out a number of experiments, and then we make a comparison of BMV and Cadence SMV. The experiments results show that for certain types of problems, BMV can perform much better than Cadence SMV in both time and memory consumption.

We believe that this is the first attempt to have an implementation of a method that combines practical error detection and verification of ACTL properties by SAT-based model checking. Similar tool can be built for LTL properties. The reason why we choose ACTL for this first attempt is that the number of iterations in the bounded model checking needed for proving or disproving ACTL properties is usually smaller than that for similar LTL properties [16], and this number is important for the practical efficiency of the method.

There are two directions for future work. One is to improve the encoding based on [12], so we can reduce the length of the encoding. The other is to use non-CNF SAT-solver like SatMate [10], thus to reduce the overhead due to the need for additional variables when converting the encoding to DIMACS format. This, however, largely depends on the performance of the non-CNF SAT solver.

References

[1] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulations. $IEEE\ Transactions\ on\ Computers$, Vol. C-35, No. 8, August, 1986: 677-691.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking:10^{20} States and Beyond. $Information\ and\ Computation$, 98(2):142-170, June 1992.

[3] S. Berezin, S. Campos and E.M. Clarke. Compositional Reasoning in Model Checking. $Proceedings\ of\ COMPOS'97$. Lecture Notes in Computer Science 1536:81-102. 1998.

[4] A. Biere, A. Cimmatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. $Proceedings\ of\ TACAS'99$, 1579, $Springer-Verlag$,193-207,1999.

[5] A. Biere, A. Cimmatti, E. Clarke, and Y. Zhu. Symbolic Model Checking Using SAT Procedures Instead Of BDDs. $Proceedings\ of\ ACM/IEEE\ Design\ Automation\ Conference(DAC'99)$,317-320,1999.

[6] E.M. Clarke, O. Grumberg, and D.A. Peled. $ModelChecking$. The MIT Press. 1999: 98-105. ISBN 0-262-03270-8.

[7] E.A. Emerson and E.M. Clarke. Using Branching-time Temporal Logics to Synthesize Synchronization Skeletons. $Science\ of\ Computer\ Programing$ 2(3):241-266. 1982.

[8] E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. $Formal\ Methods\ in\ System\ Design$9:105-131. 1995.

[9] D.S. Johnson(editor) and M.A. Trick(editor). Cliques, Coloring and Satisfiability: The Second DIMACS Implementation Challenge, vol. 26 $of\ ACM/AMS\ DIMACS\ Series$, Amer. Math. Soc., 1996.

[10] H. Jain, C. Bartzis, and E.M. Clarke. Satisfiability Checking of Non-clausal Formulas using General Matings. SAT, 2006.

[11] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. $Journal\ of\ Formal\ Methods\ in\ System\ Design$ 6:1-35.

[12] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple Bounded LTL Model Checking. $FMCAD$, 2004. 1995.

[13] K.L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. $Lecture\ Notes\ in\ Computer\ Science$ 1703:219-234. CHARME 1999.

[14] M.W. Moskewicz, C.F. Madigan, Y.Zhao, L.Zhang, and S.Malik. Chaff: Engineering an Efficient SAT Solver. $Proceedings\ of\ ACM/IEEE\ Design\ Automation\ Conference(DAC'2001)$: 530-535. 2001.

[15] D.A. Peled. Software Reliability Methods. $Springer-Verlag$. 2001.

[16] W. Penczek, B. Wozna, and A. Zbrzezny. Bounded Model Checking for the Universal Fragment of CTL. $Fundamenta\ Informaticae$ 51:135-156. 2002.

[17] V. Roy and R. de Simone. Auto/Autograph. In $Computer\ Aided\ Verification.\ DIMACS\ series\ in\ Discrete\ Mathematics\ and\ Theoretical\ Computer\ Science$ 3:235-250, June 1990.

[18] P. Wolper and P. Godefroid. Partial-Order Methods for Temporal Verification. $LNCS$ 715($CONCUR'93$):233-246. 1993.

[19] B. Wang. Proving $\forall\mu$-calculus Properties with Sat-based Model Checking. $FORTE$ 2005: 113-127.

[20] J. Zhang. Problems on the generation of finite models. $LNAI$ 814 (CADE 1994):753-757.

[21] W. Zhang. Combining Static Analysis and Case-Based Search Space Partitioning for Reducing Peak Memory in Model Checking. $Journal\ of\ Computer\ Science\ and\ Technology$ 18(6):762-770, 2003.

[22] W. Zhang. SAT-Based Verification of LTL Formulas. $FMICS$, 2006.

[23] W. Zhang. Verification of ACTL Properties by Bounded Model Checking. $EUROCAST$, 2007.

[24] http://www.Kenmcmil.com

[25] http://www.cs.chalmers.se/Cs/Research /FormalMethods/MiniSat/MiniSat.html

[26] http://fmv.jku.at/run

[27] http://nusmv.irst.itc.it/examples