Bounded Saturation Based CTL Model Checking

András Vörös and Dániel Darvas
Dept. of Measurement and Information Systems
Budapest University of Technology and Economics,
Budapest, Hungary
vori@mit.bme.hu

Tamás Bartha
Computer and Automation
Research Institute
MTA SZTAKI,
Budapest, Hungary

Abstract

Formal verification is becoming a fundamental step of safety-critical and model-based software development. As part of the verification process, model checking is one of the current advanced techniques to analyse the behaviour of a system. In this paper, we examine how the combination of two advanced model checking algorithms – namely bounded saturation and saturation based structural model checking – can be used to verify systems. Our work is the first attempt to combine these approaches, and this way we are able to handle and examine complex or even infinite state systems.

1 Introduction

Formal methods are widely used for the verification of safety critical and embedded systems. The main advantage of formal methods is that either they can provide a proof for the correct behaviour of the system, or they can prove that the system does not comply with its specification.

One of the most prevalent techniques in the field of formal verification is *model checking* [8], an automatic technique to check whether a system fulfils specification. Model checking needs a representation of the state space in order to perform analysis. Generating and storing the state space representation can be difficult in cases where the state space is very large. There are two main problems causing the state space to explode:

- independently updated state variables lead to exponential growth in the number of the system states,
- the asynchronous characteristic of distributed systems. The composite state space of asynchronous subsystems is often the Cartesian product of the local components' state spaces.

Symbolic methods [9] are advanced techniques to handle state space explosion. Instead of storing states explicitly, symbolic techniques rely on an encoded representation of the state space such as *decision diagrams*. These are compact graph representations of discrete functions.

Saturation [4] is considered as one of the most effective model checking algorithm, which combines the efficiency of symbolic methods with a special iteration strategy. Nevertheless, there are still many cases which can not be solved with the use of saturation. The state space of complex models is either too large to represent them even symbolically or their state space is infinite. Bounded model checking is an advanced technique to handle these problems, as it explores a bounded part of the state space, and examines the properties on it. Bounded saturation based state space exploration was presented in [16], where authors introduced a new saturation algorithm, which explores the state space only to some bounded depth. In this paper we extend this approach to bounded CTL model checking. Our algorithm incrementally explores the state space and employs structural model checking on it. To our best knowledge, this is the first attempt to combine saturation based CTL model checking and bounded saturation based state space exploration. Our work is a first step towards efficient bounded CTL model checking with many directions to be explored in the future.

The structure of our paper is as follows: in section 2 we introduce the background of our work. In section 3 the implemented bounded saturation algorithm is presented with our improvements. Section 4 shows the working of our bounded CTL model checking algorithm and its details. We present our measures in section 5. Related work and our conclusions come finally.

2 Background

2.1 Petri Nets

Petri nets are graphical models for concurrent and asynchronous systems, providing both structural and dynamical analysis. A (marked) discrete ordinary Petri net is a $PN = (P, T, E, w, M_0)$, represented graphically by a digraph. $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $E \subseteq (P \times T) \cup (T \times P)$ is the finite set of edges, $w : E \to \mathbb{Z}^+$ is the weight function assigning weights $w(p_i, t_j)$ to the edges between p_i and t_j. $M : P \to \mathbb{N}$ is a marking, represented by $M(p_i)$ tokens in place p_i for every i and M_0 is the initial marking of the net. A t transition is enabled, if for every incoming arc of $t : M(p_i) \geq w(p_i, t)$.

An *event* in the system is the firing of an enabled transition t_i, which decreases the number of tokens in the incoming places p_j with $w(p_j, t_i)$ and increases the number of tokens in the output places p_k with $w(t_i, p_k)$. The firing of transitions is non-deterministic. The *state space* of a Petri net is the set of states reachable through transition firings.

Figure 1(a) depicts a simple example Petri net model of a producer-consumer system. The producer creates items and places them in the buffer, from where the consumer consumes them. For synchronizing purposes the buffer's capacity is one, so the producer has to wait till the consumer takes away the item from the buffer. This Petri net model has a finite state space (also known as reachability graph) containing 8 states.

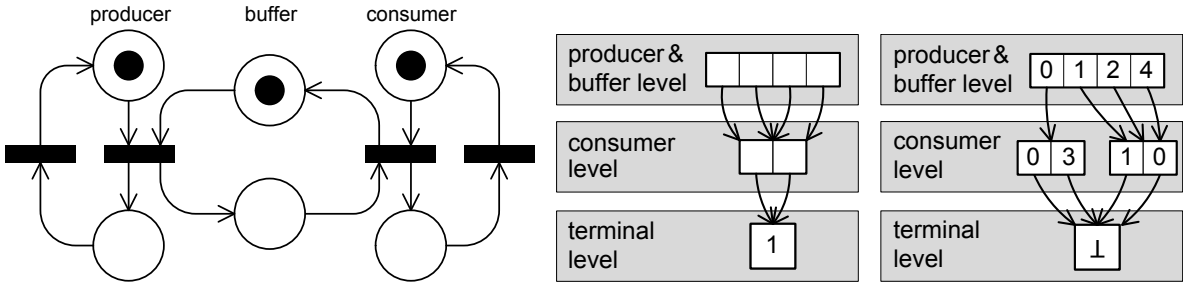
2.2 Decision Diagrams

A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f : \{0, 1, \ldots\}^K \to \{0, 1\}$. An MDD has a node set containing two types of nodes: non-terminal and two terminal nodes (0 and 1). The nodes are ordered into $K + 1$ levels. A non-terminal node is labelled by a variable index $0 < k \leq K$, which indicates to which level the node belongs (which variable it represents), and has n_k (domain size of the variable, in binary case $n_k = 2$) arcs pointing to nodes in level $k - 1$. A terminal node is labelled by the variable index 0. Duplicate nodes are not allowed, so if two nodes have identical successors in level k, they are also identical. In a quasi-reduced MDD redundant nodes are allowed: it is possible that a node's all arcs point to the same successor. These rules ensure that MDD-s are canonical and compact representation of a given function or set. The evaluation of the function is the top-down traversal of the MDD through the variable assignments represented by the arcs between nodes. Figure 1(b) depicts an MDD used for storing the encoded state space of the example Petri net. Each edge encodes a possible local state, and the possible states are the paths from the root node to the terminal *one* node.

An *Edge-valued Decision Diagram* (EDD) is an extended MDD which can represent the following f function: $f : \{0, 1, \ldots\}^K \to \mathbb{N} \cup \{\infty\}$. The differences between an MDD and an EDD are the following:

- On the terminal level there is only one terminal node, named $\perp$. This is equivalent to the terminal *one* node in an MDD.
- Every edge has a weight and a target node. We write $\langle n, w \rangle$ if the edge has weight $w \in \mathbb{N} \cup \{\infty\}$ and has target node n. In addition, we write $p[i] = \langle n, w \rangle$ if the i-th edge of the node p is $\langle n, w \rangle$ and $p[i].value \equiv w, p[i].node \equiv n$.
- If $p[i].value = \infty$, then $p[i].node = \perp$. This is equivalent to an edge in an MDD which goes to the terminal zero node.
- Every non-terminal node has an outgoing edge with weight 0.

Figure 1(c) depicts an EDD used for storing the encoded state space of the example Petri net enriched with the distance information from the initial state.



(a) The Petri net of producer-consumer model

(b) State space representation with MDD

(c) State space and state distance representation with EDD

Figure 1: Producer-consumer example

2.3 Saturation

Traditional *symbolic state space exploration* uses encoding for the traversed state space, and stores this compact, encoded representation only. Decision diagrams proved to be an efficient form of symbolic storage, as applied reduction rules provide a compact representation form. Another important advantage is that symbolic methods enable us to manipulate large set of states efficiently.

The first step of symbolic state space generation is to encode the possible states. Traditional approach encodes each state with a certain variable assignment of state variables $(v_1, v_2 \dots v_n)$, and stores it in a decision diagram. To encode the possible state changes, we have to encode the transition relation, the so called *next-state* function. This can be done in a $2n$ level decision diagram with variables: $\mathcal{N} = (v_1, v_2 \dots v_n, v'_1, v'_2 \dots v'_n)$, where the first n variables represent the *"from"*, and second n variables the *"to"* states. The next-state function represents the possibly reachable states in one step.

Usually the state space traversal builds the next-state relation during a breadth first search. The reachable set of states S from a given initial state s_0 is the *transitive closure* (in other words: the *fixed-point*) of the next-state relation: $S = \mathcal{N}^*(s_o)$. Saturation based state space exploration differs from traditional methods as it combines symbolic methods with a special iteration strategy. This strategy is proved to be very efficient for asynchronous systems modelled with Petri nets.

The saturation algorithm consists of the following steps depicted in Figure 2.

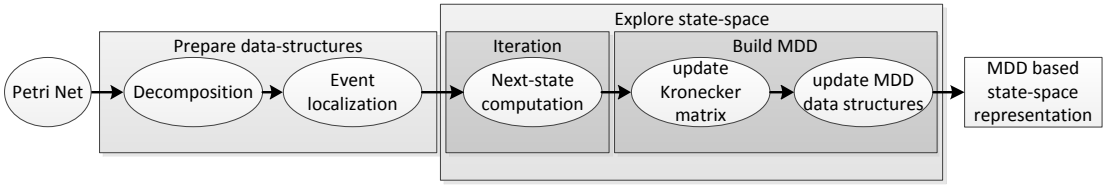


Figure 2: Saturation workflow

1. *Decomposition:* Petri nets can be decomposed into local submodels. The global state can be represented as the composition of the components' local states: $s_g = (s_1, s_2, \dots, s_n)$, where n is the number of components. This decomposition is the first step of the saturation algorithm. Ordinary

saturation needs the so-called *Kronecker consistent* decomposition [5, 6], which means that the global transition (next-state) relation is the Cartesian product of the local-state transition relations. Formally: if $\mathscr{N}_{(i,e)}$ is the next-state function of the transition (*event*) e in the i-th submodel, the global next-state of event e is: $\mathscr{N}_e = \mathscr{N}_{(1,e)} \times \mathscr{N}_{(2,e)} \times \ldots \times \mathscr{N}_{(n,e)}$. In case of asynchronous systems, a transition usually affects only some or some parts of the submodels. Event locality can be easily exploited with this decomposition. Ordinary Petri nets are Kronecker consistent for all decompositions.

2. *Event localization:* As the effects of the transitions are usually local to the component they belong to, we can omit these events from other sub-models, which makes the state space traversal more efficient. For each event e we set the border of its effect, the top (top_e) and bottom (bot_e) levels (submodels). Outside of this interval we omit the event e from the exploration.

3. *Special iteration strategy:* Saturation iterates through the MDD nodes and generates the whole state space representation using a node-to-node transitive closure. In this way saturation avoids the peak size of the MDD to be much larger than the final size, which is a critical problem in traditional approaches.

   Let $\mathscr{B}(k,p)$ represent the set of states represented by the MDD rooted at node p, at level k. Saturation applies $\mathscr{N}^*$ locally to the nodes from the bottom of the MDD to the top. Let $\mathscr{E}$ be the set of events affecting the k-th level and below, so $top_e \leq k$. We call a node p at level k saturated, iff node $\mathscr{B}(k,p) = \bigcup_{\forall e \in \mathscr{E}} \mathscr{N}_e^*(\mathscr{B}(k,p))$. The state space generation ends when the node at the top level becomes saturated, so it represents $S = \mathscr{N}^*(s_0)$.

4. *Encoding of the next-state function:* The formerly presented Kronecker consistent decomposition leads to submodels, where the next-state function can be expressed locally, with the help of the so-called Kronecker matrix: $K_{k,e}$ [3]. $K_{k,e}$ is a binary matrix and belongs to event e at level k. $K_{k,e}$ contains 1: $K_{k,e}[i,j] = 1 \Leftrightarrow j = \mathscr{N}_{k,e}(i)$. These Kronecker matrices contain only the local next-state relation. Kronecker consistent decomposition of the next-state representation turned out to be very efficient in practice.

5. *Building the MDD representation of the state space:* At first we build the MDD representing the initial state. Then we start to saturate the nodes in the first level by trying to fire all e events where $top_e = 1$. After finishing the first level, we saturate all nodes at the second level by firing all events, where $top_e = 2$. If new nodes are created at the first level by the firing, they are also saturated recursively. The procedure is continued at every level k for events, where $top_e = k$. When new nodes are created in a level below the current one, they are also recursively saturated. If the root node at the top level is saturated, the algorithm terminates. Now the MDD represents the whole state space with the next-state relation encoded in Kronecker matrices.

6. *State space representation as an MDD:* A level of the MDD generated during saturation represents the local state space of a submodel. The possible states of the submodel constitute the domain of the variables in the MDD. Each local state space is encoded in a variable.

2.4 Model Checking

Model checking is an automatic technique for verifying finite state systems. Given a model defined in Petri nets in our context, model checking decides whether the model fulfils the specification. Formally: let M be a Kripke structure (i.e. state–transition graph). Let f be a formula of temporal logic (i.e. the specification). The goal of model checking is to find all states s of M such that $M, s \models f$.

State space generation serves as a prerequisite for the structural model checking: verifying temporal properties needs the state space and transition relation representation. *CTL* (Computation Tree Logic) is widely used to express temporal specifications of systems, as it has expressive syntax and there are efficient algorithms for its analysis. Operators occur in pairs in CTL: the path quantifier, either A (on all paths) or E (there exists a path), is followed by the tense operator, one of X (next), F (future, or finally), G (globally), and U (until). However we only need to implement 3 of the 8 possible pairings due to the duality [8]: EX, EU, EG, and we can express the remaining with the help of them in the following way: $\mathsf{AX}p \equiv \neg\mathsf{EX}\neg p$, $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$, $\mathsf{AF}p \equiv \neg\mathsf{EG}\neg p$, $\mathsf{A}[p\mathsf{U}q] \equiv \neg\mathsf{E}[\neg q\ \mathsf{U}(\neg p \wedge \neg q)] \wedge \neg\mathsf{EG}\neg q$, $\mathsf{EF}p \equiv \mathsf{E}[true\ \mathsf{U}p]$. These operators also benefit from the locality exploited by saturation.

2.5 Bounded Model Checking

The main drawback of model checking is that it explores the full state space of a model. This is usually not achievable due to the high complexity of real life systems. However, we do not always need the whole state space to decide a property. Moreover, many bugs in systems belong to the category of so-called *shallow bugs*, which means that the path leading to the error is short.

Traditional model checking explores the full state space of the model so it can handle finite state systems . We can not explore the full state space of infinite systems with bounded resources. Bounded model checking gives a solution to these problems: it explores a k-bounded part of the state space in a breadth first manner, and examines the specification on this smaller part. The algorithm starts at the initial state(s) and unrolls the possible behaviours until reaching the bound. In practice, we progressively increase the bound, examining bigger and bigger parts of the state space, trying to find counterexamples or witnesses for our specification. The drawback is that if the full state space is not unrolled (i.e. the bound of the examination is chosen to be the diameter of the state space), bounded model checking is not a complete decision procedure (however nowadays there are some advanced methods which can provide completeness without reaching the state space diameter [11, 2]).

3 Bounded Saturation

Applying saturation for bounded state space generation is a difficult task: as saturation explores the state space in an irregular recursive order, bounding the recursive exploration steps does not necessarily provide this bound to be global for the state space representation. Ensuring globally bounded runs would need the change of the iteration order, to be more similar to the breadth first traversal. This way we would lose the efficiency of saturation (however the efficient symbolic representation of the state space still remains).

There are different solutions for the above problem in the literature, both for globally and locally bounded saturation based state space generation. In our work we chose one which has proved its efficiency [15]. As MDD-s provided a proper solution for state space representation, the bounded saturation algorithm needs additional distance information to be able to compute the reachability set below a bound. In order to make distance information available during the traversal, we have to store it. The algorithm in [15] used EDD-s instead of MDD-s for storing distance measures implicitly encoded into the traversed state space representation.

However this approach still differs from traditional bounded model checking approaches in the iteration order. Ordinary bounded model checking algorithms unroll the state space incrementally in breadth-first manner. Bounded saturation based state space generation algorithm follows a “fire then prune” style iteration. Bounded saturation keeps the saturation iteration order, but the algorithm fires transitions only to those local state spaces, which did not reach the bound. When bounded saturation reaches local state

Algorithm 1: BoundedEDDSaturation

output : root node : node

1 $l \leftarrow \bot$; // terminal node
2 **for** $k = 1$ **to** K **do**
3 $Confirm(k,0)$;
4 $n \leftarrow NewNode(k)$;
5 $n[0] \leftarrow \langle 0, l \rangle$;
6 $BoundedEDDSaturate(n)$;
7 $n \leftarrow CheckIn(k,n)$;
8 $l \leftarrow n$;
9 **end**
10 **return** l;

Algorithm 2: BoundedEDDSaturate

input : p : node
output : node

1 $l \leftarrow p.level$;
2 **repeat**
3 **foreach** $e \in \mathscr{E}_l$ **do** // $\forall e : top_e = l$
4 $BoundedEDDSatFire(p,e)$;
5 **end**
6 **until** *p does not change*;
7 **return** p;

Algorithm 3: BoundedEDDSatFire

input : p : node, e : event
output : changed : bool

1 $l \leftarrow p.level$;
2 $i \leftarrow 0$;
3 **if** $l < bot_e$ **then**
4 **return** $false$;
5 **end**
6 **repeat**
7 $i \leftarrow i+1$;
8 **foreach** $(i,j) \in \mathscr{N}_{l,e}$ **do**
9 **if** $p[i].value \geq bound$ **then continue**;
10 $\langle v,q \rangle \leftarrow BoundedEDDImage(p[i],e)$;
11 $\langle w,s \rangle \leftarrow Truncate(\langle v+1,q \rangle)$;
12 $\langle u,r \rangle \leftarrow UnionMin(p[j],\langle w,s \rangle)$;
13 **if** $\langle w,s \rangle \neq \langle u,r \rangle$ **then**
14 **if** *state j is not confirmed yet* **then**
15 $Confirm(l,j)$;
16 **end**
17 $p[j] = \langle u,r \rangle$;
18 **end**
19 **end**
20 **until** *p does not change*;
21 **return** $i > 1$;

Algorithm 4: BoundedEDDImage

input : $\langle v,q \rangle$: edge, e : event
output : edge

1 $l \leftarrow q.level$;
2 **if** $l < bot_e$ **then**
3 **return** $\langle v,q \rangle$;
4 **end**
5 $s \leftarrow NewNode(l)$;
6 **foreach** $(i,j) \in \mathscr{N}_{l,e}$ **do**
7 **if** $p[i].value > bound$ **then continue**;
8 $\langle v,u \rangle \leftarrow BoundedEDDImage(q[i],e)$;
9 $\langle w,o \rangle \leftarrow Truncate(\langle v,u \rangle)$;
10 $\langle u,r \rangle \leftarrow UnionMin(s[j],\langle w,o \rangle)$;
11 **if** $\langle w,o \rangle \neq \langle u,r \rangle$ **then**
12 **if** *state j is not confirmed yet* **then**
13 $Confirm(l,j)$;
14 **end**
15 $s[j] = \langle u,r \rangle$;
16 **end**
17 **end**
18 $s \leftarrow BoundedEDDSaturate(s)$;
19 $\gamma \leftarrow Normalize(s)$;
20 $s \leftarrow CheckIn(l,s)$;
21 **return** $\langle \gamma + v,s \rangle$;

Algorithm 5: TruncateApprox

input : $\langle v,p \rangle$: edge
output : edge

1 **if** $v > bound$ **then return** $\langle \infty, \bot \rangle$;
2 **else return** $\langle v,p \rangle$;

Algorithm 6: TruncateExact

input : $\langle v,p \rangle$: edge
output : edge

1 **if** $v > bound$ **then**
2 **return** $\langle \infty, \bot \rangle$;
3 **end**
4 **if** *truncate cache contains value t for p* **then**
5 **return** $\langle v,t \rangle$;
6 **end**
7 $n \leftarrow NewNode(p.level)$;
8 **foreach** $i \in S_{p.level}$ **do**
9 $r \leftarrow TruncateExact(\langle v + p[i].value, p[i].node \rangle)$;
10 $n[i] \leftarrow \langle r.value - v, r.node \rangle$;
11 **end**
12 $n \leftarrow CheckIn(p.level,n)$;
13 *insert into truncate cache n with key p* ;
14 **return** $\langle v,n \rangle$;

Figure 3: Bounded saturation algorithms

spaces in the distance from the initial state at the bound, the algorithm stops extending them any more.

Mechanism of the algorithm The bounded saturation algorithm iterates through the state space similarly to the saturation algorithm. *BoundedEDDSaturation* (Algorithm 1) starts building the state space representation in a bottom-up fashion, saturating the nodes by calling *BoundedEDDSaturate* function (Algorithm 2). *BoundedEDDSaturate* saturates the node p by firing all events e for all state i, where e is enabled: $\mathcal{N}_{k,e}(i) = j$ for some j and for this edge i: $p[i].val < bound$. $p[i].val$ is a local distance measure. By examining this edge value we only ensure that the smallest distance will not be greater than the bound. To ensure globally bounded state exploration we have to do additional computations. When reaching the distance defined by the bound, we do not explore the state space in this direction any more. After saturating a node, before stepping forward, the algorithm truncates the node in order to contain the proper bounded reachability set.

Encoding distance measure As EDD-s encode not only sets, but allow us to assign an integer value to each element of the set, it provides the ability to enrich the state space with distance measure. During the state space traversal the algorithm also builds incrementally the distance information. In *BoundedEDDSatFire* (Algorithm 3) after successful transition firing the algorithm increments the distance information encoded into the edge. This enables us to bound the state space exploration.

However, despite the fact that the algorithm prunes out the steps starting from states at the given bound, due to the irregular order of firings, we still can reach states outside the bound. These states must be cleaned out, so the algorithm uses a truncating function to cut this part of the state space representation.

Truncating excessive states There are 2 kinds of truncating functions in [15]. One which ensures exact bounded state space representation by cutting down all states belonging beyond the bound from the initial state This algorithm is *TruncateExact* (Algorithm 6). There is another truncating algorithm, which provides a coarser approximation called *TruncateApprox*, cutting only those states that exceed a local bound (Algorithm 5). This second algorithm does not necessarily remove states beyond the bound, it ensures only that states beyond a bigger bound $B \cdot K$ will be cut (K is the number of variables used in the encoding). The main difference between them is that the "exact" one computes truncating function recursively, by counting the exact distance measures. The approximate algorithm decides locally about which states to be pruned. Because of this reason, it does not have to do recursive operations, it leaves more states and needs less computations. The algorithm uses these truncating functions after the traversal of the local state spaces, before finalizing the computation of nodes. Note that our *TruncateExact* (Algorithm 6) function differs from the one presented in [15]. We reduced the computational overhead of exploring recursively the sub-MDD by using truncate cache (in *TruncateExact* algorithm in line 4 and 13). This modification significantly reduced the computational overhead making the exact truncating function competitive with the approximate one.

Our presented bounded saturation algorithm extends the former one [15] with *on-the-fly* updates of the states and the next-state relations. This way the user do not have to provide the algorithm with the possible local state spaces of the submodels, but the algorithm discovers these states and updates the transition relation according to the new information. These steps mean computational overhead, but makes the algorithm generally more usable. *Confirm(l,i)* registers at level l a new state i, and updates the transition relations with the possible next-states of state i. When local state i is confirmed, it means that it is globally reached through some firing sequences. In order to ensure consistent iteration order, the algorithm should keep transition relations up to date. Omitting these updates would lead to incomplete state space exploration. *Confirm(l,i)* is called every time when a new state is discovered: in algorithm 1 line 3, in algorithm 3 line 15 and in algorithm 4 line 13.

BoundedEDDSaturation executes the main, bottom-up saturation iteration from the initial states. By calling *Confirm* on them it is ensured, that the initial states are registered and the transition relations from the initial states are updated. It ensures that the saturation algorithm starts with the proper data structures. During the iteration the algorithm discovers new states. *BoundedEDDSatFire* and *BoundedEDDImage* discovers new states by firing transitions, their responsibility is also to update the data structures. After discovering a new state j, these functions call *Confirm(l,j)* to ensure that the algorithm continues the iteration with updated next-state relations.

4 Saturation based Bounded Model Checking

Saturation based structural CTL model checking was first presented in [7]. Later, the authors improved their algorithm, presented in [16]. In that version the authors applied a constrained saturation algorithm to prune the next-state function. Our algorithm follows a different idea: instead of exploring the whole state space and then pruning the next state function for the computation of the fixed point iterations, we restrict the state space to a given bound, and apply structural CTL model checking on this restricted part. Our approach has both advantages and disadvantages comparing to the approaches in [16]. On one hand as many modelling errors are shallow, they can be reached in a few steps, bounded model checking can find these errors efficiently. In these cases combining CTL model checking with bounded state space exploration works quite well. On the other hand, proving some properties may need the whole state space to be explored. In these cases bounded model checking can not give a proper answer unless the bound is chosen to be the diameter of the state space. In these cases bounded model checking usually means overhead, and the main factor become the efficiency of the fixed-point iterations. In the following we introduce how MDD data structures and saturation can help CTL model checking, and after it we present the bounded CTL model checking algorithm.

CTL model checking The CTL model checking algorithm efficiently exploits the formerly (during the state space exploration) created data structures. As CTL operators express next-state relations and fixed point properties, we have to efficiently express the inverse of the next-state function $\mathcal{N}^{-1}$. The semantics of the 3 implemented CTL operators:

- **EX**: $i^0 \vDash EXp$ iff $\exists i^1 \in \mathcal{N}(i^0)$ s.t. $i1 \vDash p$ ("$\vDash$" means "satisfies"). This means that EX corresponds to the inverse $\mathcal{N}$ function, applying one step backward through the next-state relation. Saturation computes this step-back by transposed Kronecker matrix. This way we can exploit locality efficiently.

- **EG**: $i^0 \vDash EGp$ iff $\forall n \geq 0, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash p$ so that there is a strongly connected component containing states satisfying p. This computation needs a greatest fixed-point computation, so that saturation cannot be applied directly to it. Computing the closure of this relation however benefits from the locality accompanying the decomposition.

- **EU**: $i^0 \vDash E[pUq]$ iff $\exists n \geq 0, \exists i^1 \in \mathcal{N}(i^0), \ldots, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash q$ and $i^m \vDash p$ for all $m < n$. Informally: we search for a state q reached through only states satisfying p. The computation of this property needs a least fixed-point computation, which can exploit the efficiency of saturation.

Saturation builds Kronecker matrix based next-state representations: this makes the building of the inverse relation easy as if $\mathcal{N}_e = \mathcal{N}_{(1,e)} \times \mathcal{N}_{(2,e)} \times \ldots \times \mathcal{N}_{(n,e)}$, then $\mathcal{N}_e^{-1} = \mathcal{N}_{(1,e)}^{-1} \times \mathcal{N}_{(2,e)}^{-1} \times \ldots \times \mathcal{N}_{(n,e)}^{-1}$, where $\mathcal{N}_{(k,e)}^{-1}$ is the inverse next-state relation of $\mathcal{N}_{(k,e)}$ $\forall k \in 1 \ldots n$. This inverse next-state relation

is expressed by a Kronecker matrix: if $K_{(k,e)}$ is the Kronecker representation of $\mathcal{N}_{(k,e)}$, then $\mathcal{N}_{(k,e)}^{-1}$ is expressed with $K_{(k,e)}^T$, where K^T is the transposed matrix of K.

Before performing saturation in EU, we have to classify events into categories in order to define the breadth first and the saturation based steps in the fixed-point calculation. The algorithm needs this classification because saturation can be applied only to those events, which can not lead the path out of p. We need this constraint because of the irregular order in which saturation explores states.

- An event e is *dead* with respect to a set of states S if $\mathcal{N}_e^{-1}(S) \cap S = \emptyset$. These events are omitted from the fixed-point calculation.
- An event e is *safe*, if it cannot lead from outside S to sates in S, formally: $\emptyset \subset \mathcal{N}_e^{-1}(S) \subseteq S$.
- All other events are *unsafe*.

With the help of this categorization, we decompose the fixed-point calculation into 2 steps:

- Computing the closure of relations of the *safe* events can be efficiently done by saturation.
- By breadth-first traversal the algorithm explores *unsafe* events.

As from states reached by unsafe steps we have to filter out those, which do not satisfy p or q, we have to compute the intersection of them with $p \cup q$. This intersection is evaluated in all iteration. The efficiency of EU computation highly depends on the efficiency of the saturation steps. The number of breadth first steps (and intersection operations) depends on the model and the temporal logic formula itself. Saturation makes only the exploration of the safe part more efficient.

Bounded CTL model checking algorithm Our approach presented in this paper is the first which combines saturation based model checking with bounded saturation based state space traversal. Our approach has many advantages compared to the traditional structural model checking algorithm, however there are still many directions to improve it.

Bounded model checking is started with input values: *initial bound (B)*, which is the first bound where the algorithm stops and executes model checking, the *specification* to be examined on the *Petri net* model, and an *increment (δ)*, which is used to compute the next bound. At first, the algorithm unfolds the state space to the distance of the *initial bound ($b = B$)*. The output of the state space generation is the state space encoded in an EDD. Before starting the CTL model checking, the EDD is converted into MDD by throwing away the distance information and applying the MDD reduction rules. The main advantage of converting the EDD to MDD is that MDD-s are usually more compact than EDD-s (note that the cause of this compaction is not only the smaller data structures needed to represent edges but applied reduction rules may merge more nodes than in EDD-s). Since CTL model checking is usually a memory intensive task, it is important to apply as compact state space representation as possible. Saturation based CTL model checking algorithm is executed on the bounded state space representation MDD. The result of the model checking can trigger the algorithm to stop, if the result of the model checking is the expected. Otherwise we have to examine if the state space diameter is reached (i.e the full state space is explored). If the result is not the expected, and the full state space is not even discovered, the algorithm continues running with a new bound $b = b + \delta$

The main advantage of this method is that the algorithm supports the full CTL semantic, which is usually not the case for traditional bounded model checkers. This is complementary to the widespread automaton theoretic approach, where the examined properties are expressed with the help of linear temporal logic, and the model checker unfolds the automaton till the given bound, to examine all possible behaviours. Our structural model checking algorithm follows a completely different approach.

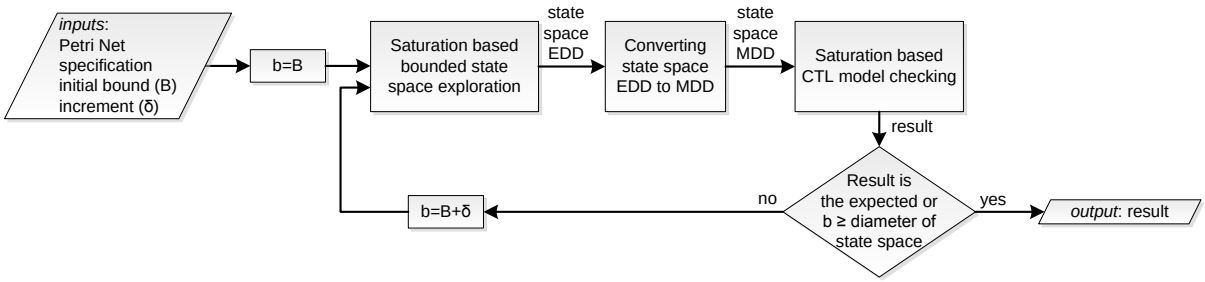


Figure 4: Bounded model checking process

5 Evaluation

Our algorithm is the first which combines bounded saturation based state space exploration with saturation based CTL model checking. We have developed an experimental implementation of our algorithm in the C# programming language. Our main purpose was to examine the efficiency of our algorithm and compare it to the classical CTL model checking. We also examine how can saturation based bounded state space traversal make CTL based model checking more scalable. We have implemented the saturation based CTL model checking algorithms from [7] on top of the bounded saturation based state space generation. In this section we compare our bounded model checking algorithm to its classical counterpart from [7], which we also implemented. Our experiments confirm what many former research found: bounded and classical model checking are complementary techniques. Bounded model checking can efficiently find errors in the vicinity of the initial states. Classical model checking is more efficient for those problems, which need the examination of a bigger part of the state space.

We used a desktop PC for the measurements: Intel Core2 Quad CPU Q8400 2.66 GHz CPU, 4 GB memory with Windows 7 Enterprise and .NET 4.0 x64.

The models we used for the evaluation are widely known in the model checking community. Flexible Manufacturing System (FMS) and Kanban system are models of production systems [4]. The parameter N refers to the complexity of the model and it influences the number of the tokens in it. Their state space scales from 10^{20} to 10^{30} states. Slotted Ring (SR) is a model of communication protocol [14], where N is the number of participants in the communication. The size of the state space of the SR-100 model is about 10^{100}. We also used a model of Hanoi towers from [13]. The state space of our Hanoi towers model with 12 rings is 531 441 states. We exploited the expressive power of Petri nets with inhibitor arcs in order to have more concise models. Our models are provided in PNML format, and they can be downloaded from our homepage [17]. Our algorithm can be fine tuned by the user as both the initial bound and the increment distance can be defined by the user. This way when we wait that we need to explore a smaller part of the state space, the algorithm can be set to be optimal for smaller distances. However it is also possible that we choose both the initial bound and the increment distance bigger to find a proof in fewer iterations when we guess that the property is "deeper". Some knowledge about the expected behaviour of the property can significantly reduce the computational time.

In Table 1 we compare bounded model checking with the classical approach. We have done measurements with both the *Approximative* and *Exact* truncating function. In Table 1 it can be seen that for some properties, our bounded model checking approach is more efficient than the classical one, for some models and temporal properties by an order of magnitude. It can also be seen that for Hanoi model (which has less concurrency) by using the *Exact* truncating function we can reduce the runtime comparing to the *Approximative* algorithm. It is surprising as former researches ([15]) stated the contrary. However, we hope that our cache based optimized truncating function is the main reason for being competitive with

the *Approximative* algorithm. For the other models *Approximative* truncating function performs better, and this is true generally. The main reason why bounded model checking performs well is that for these models the given specification property could be proven in bound always less than 128 (for Hanoi towers). The diameter of their state space is usually much longer: it is 4096 in the case of Hanoi tower, 420 steps in the case of Kanban model and scales from 320 to 2800 steps in the case of FMS model.

In the Table 2 we compare the bounded model checking algorithm to the classical CTL model checking algorithm, and we examine how the longer distance to reach a proof effects the runtimes of them. Bounded model checking works well for the first case, because we have to take 10 steps to reach a proof. However, as the needed number of steps increased, the runtimes grow too. Contrary, classical model checking algorithm always needed the same time to prove the property.

Last table (Table 3) depicts how the decreasing number of main iterations effects the runtime. If the algorithm increases the bound with bigger "increments", the algorithm finds a proof earlier, reducing the overhead caused by the unsuccessful bounded model checking steps.

Model	**Expression**	**Approx**	**Exact**	**Classic**
Hanoi-12	$EF(B_4 > 0)$	13,23 s	11,56 s	31,33 s
Hanoi-12	$EF(B_5 > 0)$	1,93 s	1,76 s	31,27 s
Hanoi-12	$EF(B_6 > 0)$	0,41 s	0,37 s	31,33 s
Hanoi-12	$EF(B_8 > 0)$	0,03 s	0,03 s	31,33 s
SlottedRing-100	$EF(E_2 = 1 \wedge A_2 = 1)$	0,39 s	0,80 s	11,23 s
Kanban-30	$EG(true)$	0,05 s	0,03 s	21,56 s
Kanban-30	$EF(P_{out4} = 1)$	0,00 s	0,02 s	19,89 s
Kanban-30	$EF(P_{out4} = 5)$	0,90 s	3,54 s	19,88 s
Kanban-30	$EF(P_{out4} = 10)$	8,61 s	109,64 s	19,89 s
FMS-25	$EG(true)$	0,08 s	0,17 s	0,53 s
FMS-100	$EG(true)$	0,05 s	0,17 s	20,32 s
FMS-200	$EG(true)$	0,06 s	0,16 s	209,46 s

Table 1: Runtime results of the algorithms

Model	**Expression**	**Approx**	**Classic**
FMS-100	$EF(P1s = 10)$	3,82 s	12,24 s
FMS-100	$EF(P1s = 20)$	24,37 s	12,20 s
FMS-100	$EF(P1s = 50)$	147,12 s	12,09 s
FMS-100	$EF(P1s = 100)$	512,27 s	12,02 s

Table 2: Scaling of the runtime with the growing number of necessary steps

Model	**Incr.**	**Approx**	**Exact**	**Classic**
Hanoi-12	10	6,65 s	7,74 s	31,33 s
Hanoi-12	20	3,49 s	3,76 s	31,33 s
Hanoi-12	30	2,57 s	2,62 s	31,33 s
Hanoi-12	40	2,23 s	2,59 s	31,33 s

Table 3: Scaling of the runtimes with increasing increments (expression: $\mathsf{EF}(B_4 > 0)$)

6 Related Work

Saturation based bounded state space exploration was first presented in [15]. This algorithm served as the base of our bounded model checking algorithm. Saturation based CTL model checking was presented in [7]. Bounded model checking is a wide-spread technique in the field of hardware verification, for further details see for example: [2, 11, 1].

SAT based bounded approaches proved their efficiency in hardware verification, but there are also efficient techniques for the examination of Petri nets. We refer the reader to [10, 12].

7 Conclusion and future work

We have presented a combined approach to model check Petri nets. This presentation is the first about our results, and it suggests that there is much work left. At first, we would like to build incrementally the state space after each iteration. The main problem now is that truncating the state space invalidates many of the used caches and data structures. Because of this reason, we always start to build the state space representation from scratch. However, by applying a simple transformation, and in the price of losing some distance information, we could exploit the benefits of incremental state space construction. We hope that it would significantly reduce the runtime for bigger models. We also plan to use the so called constrained saturation to make the CTL model checking more efficient. However much work is left, we think that we could prove the usability of our algorithm, and we could combine the the efficiency of saturation with bounded model checking.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs, 1999.

[2] A. R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. on Computing*, 12:203–222, July 2000.

[4] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2031*, pages 328–342. Springer-Verlag, 2001.

[5] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state-space exploration. *Int. J. Softw. Tools Technol. Transf.*, 8(1):4–25, 2006.

[6] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In *Proceedings of the 9th ICCPE: Modelling Techniques and Tools*, pages 44–57, London, UK, 1997. Springer-Verlag.

[7] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification (CAV'03), LNCS 2725*, pages 40–53. Springer-Verlag, 2003.

[8] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[9] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*. 1996.

[10] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory*, CONCUR '01, pages 218–232, London, UK, 2001. Springer-Verlag.

[11] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.

[12] S. Ogata, T. Tsuchiya, and T. Kikuno. SAT-based verification of safe petri nets. In *ATVA'04*, 2004.

[13] R. Saad, S. Dal Zilio, and B. Berthomieu. Mixed shared-distributed hash tables approaches for parallel state space construction. In *ISPDC*, Cluj Napoca, Romania, 07/2011 2011. IEEE.

[14] A. Vörös, T. Bartha, D. Darvas, T. Szabó, A. Jámbor, and Á. Horváth. Parallel saturation based model checking. In *ISPDC*, Cluj Napoca, 2011. IEEE Computer Society, IEEE Computer Society.

[15] A. Yu, G. Ciardo, and G. Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *Int. J. Softw. Tools Technol. Transf.*, 11:117–131, February 2009.

[16] Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pages 368–381, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Homepage of the *PetriDotNet* framework. `http://petridotnet.inf.mit.bme.hu/`. [Online; accessed 31-Aug-2011].