Proceedings of the Estonian Academy of Sciences,
2013, **62**, 1, 59–70
doi: 10.3176/proc.2013.1.07
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

Bounded saturation-based CTL model checking

András Vörös[a*], Dániel Darvas[a], and Tamás Bartha[b]

[a] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Magyar tudósok körútja 2., H-1117 Budapest, Hungary
[b] Computer and Automation Research Institute, Hungarian Academy of Sciences, Kende u. 13–17., H-1111 Budapest, Hungary

Received 31 August 2011, revised 10 April 2012, accepted 1 November 2012, available online 20 February 2013

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. Symbolic model checking is an efficient approach to handling even complex models with huge state spaces. Saturation is a symbolic algorithm with a special iteration strategy, which is efficient for asynchronous models. Recent advances have resulted in many new kinds of saturation-based algorithms for state space generation and bounded state space generation and also for structural model checking. In this paper, we examine how the combination of two advanced model checking algorithms – 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. Our measurements show that we can exploit the efficiency of saturation in bounded model checking.

Key words: bounded model checking, saturation, Multiple-valued Decision Diagram, temporal logic, Computation Tree Logic.

1. INTRODUCTION

Formal methods are becoming widely used for the verification of safety-critical and embedded systems. The main advantage of formal methods is that (when applicable) they can either 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* [9], an automated technique to check whether a system fulfils its 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 when the state space is very large. There are two main problems that cause the state space to explode:

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

Symbolic methods [10] 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 forms of discrete functions.

Saturation [5] is considered as one of the most effective state space exploration algorithms, which combines the efficiency of symbolic methods with a special iteration strategy. Nevertheless, there are still many cases in which the state space of complex models is either too large to store even symbolically, or the state space is infinite. *Bounded model checking* is an advanced technique to handle these problems, as it explores and examines the properties on a bounded part of the state space.

Bounded saturation-based state space exploration was presented in [20], where the authors introduced a new saturation algorithm, which explores the state space only to some bounded depth. In this paper we extend their approach to bounded Computation Tree Logic (CTL) model checking. Our algorithm incrementally explores the state space and employs structural model checking on it. To our best knowledge, this is the

*Corresponding author, vori@mit.bme.hu

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 described with our improvements. Section 4 describes the operation of our bounded CTL model checking algorithm and its details. We present our measurements in Section 5. Finally, we summarize the related work and give our conclusions and directions for future work.

2. BACKGROUND

In this section we outline the background of our work. First, we present Petri nets, the modelling formalism we used. Then we describe decision diagrams, in particular the Multiple-valued Decision Diagrams and Edge-valued Decision Diagrams. These are the underlying data structures of our algorithms; they store the state space during model checking. Finally, we present the saturation-based state space exploration algorithm, the model checking background, and the use of saturation for bounded state space exploration.

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. $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \dots, t_m\}$ 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 1a 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 eight states.

2.2. Decision diagrams

A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f : \{0, 1, \dots\}^K \to \{0, 1\}$. 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 $0 < k \leq K$ that 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 at 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 at 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 an MDD is a 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 1b 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 that can represent the following f function: $f : \{0, 1, \dots\}^K \to \mathbb{N} \cup \{\infty\}$. The differences between an MDD and an EDD are the following:

- At 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 ith 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 that goes to the terminal zero node.
- Every non-terminal node has an outgoing edge with weight 0.

Figure 1c depicts an EDD storing the encoded state space enriched with the distance information (computed from the initial state). Every p node is visualized as a rectangle with k slots, where k is the number of children (domain of the variable). The ith edge starts from the ith slot of the p node, and the value $p[i].value$ (the weight of the edge) is written to that slot. Usually the zero valued dangling edges and the ∞ valued edges are not shown.

In the example of Fig. 1c let the node on the left side of the *consumer level* be x. This x node has two children: $x[0] = \langle \perp, 0 \rangle$ and $x[1] = \langle \perp, 3 \rangle$.

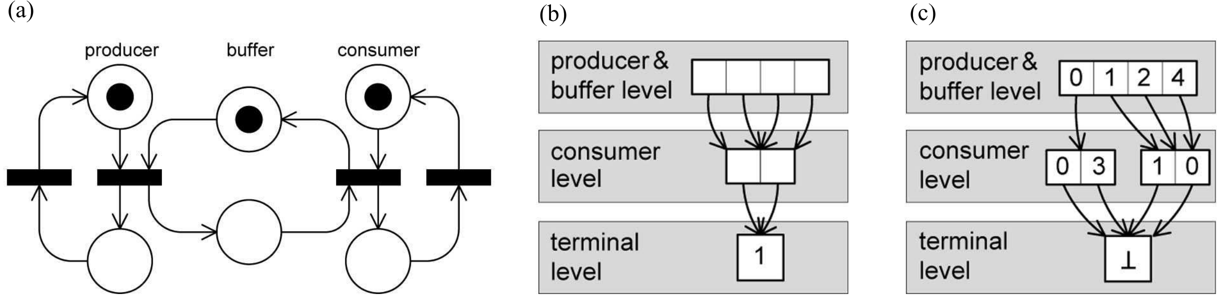

Fig. 1. Example of a producer–consumer system: (a) the Petri net of the producer–consumer model, (b) state space representation with an MDD, and (c) state space and state distance representation with an EDD.

2.3. Saturation

Traditional *symbolic state space exploration* uses encoding for the traversed state space and stores this compact, encoded representation only. Decision diagrams have 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 sets of states efficiently.

The first step of symbolic state space generation is to encode the reachable states. The 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. The transition relation, the so-called *next-state* function, needs to be encoded in order to encode the possible state changes. 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 the second n variables the "*to*" states. The next-state function represents the 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 $S = \mathcal{N}^*(s_0)$. Saturation-based state space exploration differs from the traditional methods, as it combines symbolic methods with a special iteration strategy. This strategy proved to be very efficient for asynchronous systems modelled with Petri nets.

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

1. *Decomposition*: Petri nets can be decomposed into local submodels. The global state can be represented as the composition of the local states of components $s_g = (s_1, s_2, \dots, s_n)$, 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 [6,7], which means that the global transition (next-state) relation is the Cartesian product of the local-state transition relations.

   Formally: if $\mathcal{N}_{(i,e)}$ is the next-state function of the transition (*event*) e in the ith submodel, the global next-state of event e is $\mathcal{N}_e = \mathcal{N}_{(1,e)} \times \mathcal{N}_{(2,e)} \times \dots \times \mathcal{N}_{(n,e)}$. 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 submodels, 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 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 $\mathcal{B}(k, p)$ denote the set of states represented by the MDD rooted at node p, at level k. Saturation

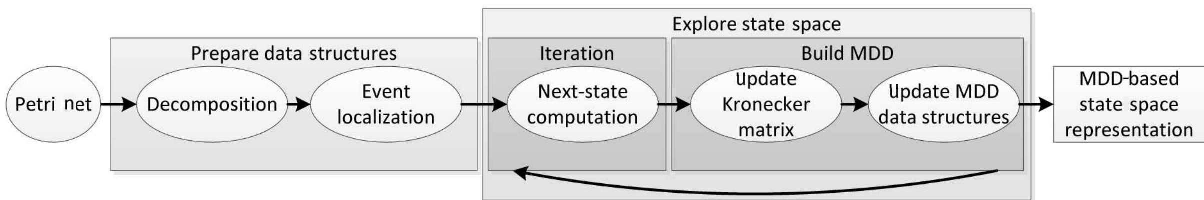

Fig. 2. Saturation workflow.

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

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: $K_{k,e}$ [4]. $K_{k,e}$ is a binary matrix and belongs to event e at level k. $K_{k,e}$ contains 1: $K_{k,e}[i,j] = 1 \Leftrightarrow j = \mathcal{N}_{k,e}(i)$. 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*: First we build the MDD representing the initial state. Then we start to saturate the nodes at 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 will also be saturated recursively. The procedure is continued at every level k for events where $top_e = k$. When new nodes are created at a level below the current one, they will also be recursively saturated. If the root node at the top level is saturated, the algorithm will terminate. 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 automated 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., a state-transition graph). Let f be a requirement expressed as a temporal logic formula (i.e., the specification). The goal of model checking is to find all states s of M such that $M, s \vDash f$.

State space generation serves as a prerequisite for the structural model checking: verifying temporal properties needs the state-space and transition-relation representation. CTL is widely used to express temporal specifications of systems, as it has expressive syntax and efficient analysis algorithms are available for it. In CTL operators occur in pairs: 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 three of the eight possible pairings due to the duality [9] EX, EU, EG, and the remaining quantifier–operator pairings can be expressed with the help of these three 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 expressions also benefit from the locality exploited by saturation.

2.5. Bounded model checking

The main drawback of model checking is that it needs to explore the full state space of a model. In practice this is not always achievable due to the high complexity of real-life systems. However, on every occasion it is not necessary to analyse the whole state space to decide a property. Moreover, many design and implementation errors in systems are so-called *shallow bugs*, meaning that the path leading to the error is short.

Traditional model checking explores the full state space of the model. Therefore it can only handle finite state systems, as the full state space of infinite systems cannot be explored with finite resources. Bounded model checking gives a solution: it explores a finite, k-bounded depth part of the state space in a breadth-first manner and examines the specification in this smaller part.

The algorithm starts at the initial state(s) and traverses the possible trajectories until it reaches the bound. The main idea is to progressively increase the bound, examining larger and larger parts of the state space, trying to find counterexamples or witnesses for the requirements. The drawback of this approach is that if the full state space is not unrolled (i.e., the bound of the traversal is chosen to be less than the diameter of the state space), bounded model checking will not provide a complete decision procedure (although nowadays some advanced methods can guarantee completeness without reaching the state space diameter [2,13]).

3. BOUNDED SATURATION

Applying saturation for bounded state space generation is a difficult task: since saturation explores the state space in an irregular recursive order, bounding the recursive exploration steps does not necessarily guarantee this bound to be global for the whole state space representation. In order to ensure globally bounded trajectories, the iteration order would need to be made more similar to the breadth-first traversal. This, however, would lead to losing the efficiency of saturation (although the resulting compact symbolic representation of the state space is still an advantage).

In the literature there are different solutions for the above problem both for globally and locally bounded

saturation-based state space generation. In our work we chose the one that has already proved its efficiency [19]. The bounded saturation algorithm needs additional information about traversal distance to be able to compute the reachability set below a bound. MDDs are a highly efficient storage form for state space representation, but they do not include this information. In order to make the distance information available during the traversal, the algorithm in [19] uses EDDs instead of MDDs for storing distance measures implicitly encoded into the state space representation.

Traditional bounded model checking algorithms explore the state space incrementally in breadth-first manner. Bounded saturation-based state space generation follows a "fire then prune" style iteration. Bounded saturation keeps the same iteration order as saturation, but the algorithm fires only transitions leading to local state spaces that do not reach the bound. When bounded saturation unrolls a local state space having a distance from the initial state equal to the bound, the algorithm will not extend it any further.

3.1. Implementation of bounded saturation

The bounded saturation algorithm (Fig. 3) 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 states 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. Examining this edge value only ensures that the smallest distance will not be greater than the bound. Additional computation is necessary to implement globally bounded state exploration. When the distance defined by the bound is reached, the state space is not explored in this direction any more. After saturating a node, and before stepping forward, the algorithm truncates the node in order to contain the proper bounded reachability set.

Encoding the distance measure. EDDs allow assigning an integer value to each element of the set they encode, providing the ability to supplement the state space with a distance measure. During state space traversal the algorithm also updates this distance information incrementally. In *BoundedEDDSatFire* (Algorithm 3) the algorithm increments the distance information encoded into the edge after successfully firing a transition. This enables bounding the state space exploration.

However, despite the fact that the algorithm prunes out steps subsequent to states located at the given bound, some states located outside the bound can still be reached due to the irregular order of the firings. These states must be avoided; therefore the algorithm uses a truncating function to omit this part of the state space representation.

Truncating excessive states. There are two types of truncating functions in [19]. The algorithm uses these truncating functions after the traversal of the local state spaces, before finalizing the computation of nodes.

1. The first one ensures exact bounded state space representation by excluding all states located beyond the bound measured from the initial state. This algorithm is *TruncateExact* (Algorithm 6).
2. The other one provides a coarser approximation called *TruncateApprox*, excluding only those states that exceed a local bound (Algorithm 5). This algorithm does not necessarily remove states beyond the bound, it only ensures that states beyond a larger bound $B \cdot K$ will be excluded (where K is the number of variables in the encoding).

The main difference between the two is that the "exact" method computes the truncating function recursively, by counting the exact distance measures. The approximate algorithm decides locally which states to prune. Consequently, it does not have to do recursive operations, leaving more states and needing less computation.

Our approach uses an enhanced version of the *TruncateExact* (Algorithm 6) function that differs from the one presented in [19]. We reduced the computational overhead of exploring recursively the sub-MDD by using a truncate cache (in the *TruncateExact* algorithm see lines 4 and 13). This modification reduced the computational overhead significantly, making the "exact" truncating function competitive with the approximate one.

Our bounded saturation algorithm extends the former one [19] with *on-the-fly updates* of the states and the next-state relations. This way the user does not have to provide the algorithm with the possible local state spaces of the submodels. The algorithm itself discovers these states and updates the transition relation according to the new information instead. The extra steps add some computational overhead, but improve the usability of the algorithm in general.

Confirm(l,i) registers a new state i at level l and updates the transition relations with the possible next states of state i. When a local state i is confirmed, this means that the state is globally reachable through some firing sequences. In order to ensure a consistent iteration order, the algorithm must 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. Calling *Confirm* ensures that the initial states are registered and the transition relations from the initial states are updated. Consequently, the saturation algorithm starts with the proper data structures. During the iteration the algorithm calls *BoundedEDDSatFire* and *BoundedEDDImage* to discover new states by firing transitions. These functions are also responsible for updating the data structures. After discovering a new state j, they call *Confirm(l,j)* to make sure that the algorithm continues the iteration with updated next-state relations.

Algorithm 1: *BoundedEDDSaturation*

```
   output : root node : node
1  l ← ⊥ ;                        // terminal node
2  for k = 1 to K do
3      Confirm(k,0) ;
4      n ← NewNode(k);
5      n[0] ← ⟨0,l⟩;
6      BoundedEDDSaturate(n);
7      n ← CheckIn(k,n);
8      l ← n;
9  end
10 return l;
```

Algorithm 2: *BoundedEDDSaturate*

```
   input  : p : node
   output : node
1  l ← p.level;
2  repeat
3      foreach e ∈ E_l do               // ∀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 ← p.level;
2  i ← 0;
3  if l < bot_e then
4      return false;
5  end
6  repeat
7      i ← i+1;
8      foreach (i, j) ∈ N_{l,e} do
9          if p[i].value ≥ bound then continue;
10         ⟨v,q⟩ ← BoundedEDDImage(p[i],e);
11         ⟨w,s⟩ ← Truncate(⟨v+1,q⟩);
12         ⟨u,r⟩ ← UnionMin(p[j],⟨w,s⟩);
13         if ⟨w,s⟩ ≠ ⟨u,r⟩ then
14             if state j is not confirmed yet then
15                 Confirm(l, j) ;
16             end
17             p[j] = ⟨u,r⟩;
18         end
19     end
20 until p does not change;
21 return i > 1;
```

Algorithm 4: *BoundedEDDImage*

```
   input  : ⟨v,q⟩ : edge, e : event
   output : edge
1  l ← q.level;
2  if l < bot_e then
3      return ⟨v,q⟩;
4  end
5  s ← NewNode(l);
6  foreach (i, j) ∈ N_{l,e} do
7      if p[i].value > bound then continue;
8      ⟨y,a⟩ ← BoundedEDDImage(q[i],e);
9      ⟨w,o⟩ ← Truncate(⟨y,a⟩);
10     ⟨u,r⟩ ← UnionMin(s[j],⟨w,o⟩);
11     if ⟨w,o⟩ ≠ ⟨u,r⟩ then
12         if state j is not confirmed yet then
13             Confirm(l, j) ;
14         end
15         s[j] = ⟨u,r⟩;
16     end
17 end
18 s ← BoundedEDDSaturate(s);
19 γ ← Normalize(s);
20 s ← CheckIn(l,s);
21 return ⟨γ+v,s⟩;
```

Algorithm 5: *TruncateApprox*

```
   input  : ⟨v, p⟩ : edge
   output : edge
1  if v > bound then return ⟨∞,⊥⟩;
2  else return ⟨v, p⟩;
```

Algorithm 6: *TruncateExact*

```
   input  : ⟨v, p⟩ : edge
   output : edge
1  if v > bound then
2      return ⟨∞,⊥⟩;
3  end
4  if truncate cache contains value t for ⟨v, p⟩ then
5      return ⟨v,t⟩;
6  end
7  n ← NewNode(p.level);
8  foreach i ∈ S_{p.level} do
9      r ← TruncateExact(⟨v + p[i].value, p[i].node⟩);
10     n[i] ← ⟨r.value − v,r.node⟩;
11 end
12 n ← CheckIn(p.level,n);
13 insert into truncate cache n with key ⟨v, p⟩ ;
14 return ⟨v,n⟩;
```

Fig. 3. Bounded saturation algorithms.

4. SATURATION-BASED BOUNDED MODEL CHECKING

Saturation-based structural CTL model checking was introduced in [8]. Later, the algorithm was improved in [20]. In the latter publication 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 compared to the approach in [20]. On the one hand, many design and implementation errors are shallow, thus they can be reached in a few steps, and so 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 (for example invariant properties) may need the whole state space to be explored. In these cases bounded model checking cannot give a proper answer unless the bound is chosen to be the diameter of the state space. Using bounded model checking for such problems often means overhead, and the efficiency of the fixed-point iterations becomes the main performance factor. It is usually difficult to define the bound where the algorithm explored a sufficiently large part of the state space to decide about the specification. In our work we extended the model checking framework with three-valued logic to support decision-making. We present this extension at the end of this section.

In the following we demonstrate how MDD data structures and saturation can help CTL model checking. After that we present our bounded CTL model checking algorithm.

4.1. CTL model checking

The CTL model checking algorithm [8] can utilize well the data structures created during the state space exploration. CTL operators express next-state relations and fixed-point properties, thus we have to efficiently compute the inverse of the next-state function $\mathcal{N}^{-1}$.

The semantics of the three implemented CTL operators is the following:

- **EX**: $i^0 \vDash \mathsf{EX}\, p$ iff $\exists i^1 \in \mathcal{N}(i^0)$ s.t. $i^1 \vDash p$ (where $\vDash$ means “satisfies”). Consequently, EX corresponds to the inverse $\mathcal{N}$ function, moving one step backward through the next-state relation. Saturation computes this back-step by a transposed Kronecker matrix. This way we can take advantage of the locality.
- **EG**: $i^0 \vDash \mathsf{EG}\, p$ 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, thus saturation cannot be applied directly. Nevertheless, computing the closure of this relation profits from the locality accompanying the decomposition.
- **EU**: $i^0 \vDash \mathsf{E}[p \mathsf{U} q]$ 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 solely through states satisfying p. The computation of this property needs a least fixed-point computation, which can benefit from the efficiency of saturation.

Saturation builds Kronecker matrix-based next-state representations. This makes the building of the inverse relation easy, since 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 the case of the EU operator, 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 that do 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 states 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 two steps:

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

We have to filter out those states reached by unsafe steps that do not satisfy p or q by computing the intersection of $p \cup q$. This intersection is evaluated in every iteration. The efficiency of EU computation depends highly 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 more efficient only the exploration of the safe part.

4.2. Bounded CTL model checking algorithm

Our approach presented in this paper is the first that combines saturation-based model checking with bounded saturation-based state space traversal (Fig. 4). It has many advantages compared to the traditional structural model checking algorithms; however, there are still many directions to improve it.

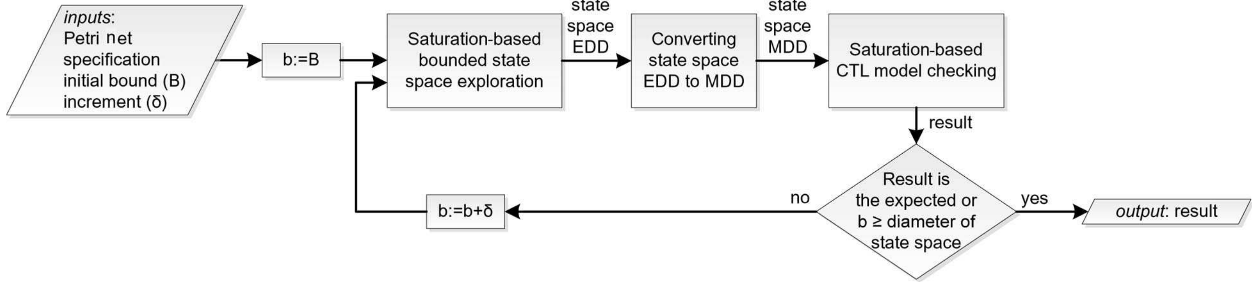

Fig. 4. Bounded model checking process.

Bounded model checking has three input parameters:

- the *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.

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 an MDD by throwing away the distance information and applying the MDD reduction rules. The main advantage of converting the EDD to an MDD is that MDDs are usually more compact than EDDs. (Note that the reason for this compaction is not only the smaller data structures needed to represent edges, but also that the applied reduction rules may merge more nodes than in EDDs.) Since CTL model checking is usually a memory-intensive task, it is important to make the state space representation as compact as possible.

The saturation-based CTL model checking algorithm is executed on the bounded state space representation MDD. If the result of the model checking is equal to that expected, the algorithm will stop. Otherwise we have to check if the state space diameter is reached (i.e., the full state space has been explored). If neither the result is the expected one nor the full state space has been discovered yet, the algorithm will continue running with an increased bound $b := b + \delta$.

The main advantage of this approach is that the analysis supports the full CTL semantics, which is usually not the case for traditional bounded model checkers. This is complementary to the wide-spread automaton theoretical 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 in order to examine all possible behaviours. Our structural model checking algorithm follows a completely different approach.

4.3. Decision-making with three-valued logic

Classical model checking uses two-valued logic, where the set of states is categorized according to the formula into two sets: those states from which the property is fulfilled and those states from which the property does not hold. Evaluating a formula in CTL model checking returns a set of states satisfying the formula. In the case of nested CTL operators, the result of the nested formula is the input for the outer formula.

The problem with bounded model checking is that essentially it is a *semi-decision procedure*, thus it can also produce an *unknown* result about a given property. In our work we extended our former saturation-based bounded model checking approach to handle uncertainty about the specification, so it can produce three results: *true*, *false*, or $\perp$ (where $\perp$ denotes the unknown result). *Three-valued logic* [12,17] is a mathematical reasoning procedure for these three values, applied in many areas, including the analysis of asynchronous circuits, compilers, and model checking [3,16]. First we have to extend the logic, which can be seen in Table 1.

4.3.1. CTL semantics with three-valued logic

Our model checking algorithm computes the set of states S_{true} of the model M for formula f, where $S_{true} = \{s_0 : M, s_0 \models f, s_0 \in S\}$. All other states are in the set S_{false} as for them the specification, i.e., the CTL formula, evaluates to *false*. In the following we consider a CTL expression c as a function assigning a truth value to each state in the state space $c : S \rightarrow \mathbb{B}$, where S is the set of all states in the model. The traditional model checking question is to examine if the initial state is in the states fulfilling the specification $s_0 \subseteq S_{true}$ or not (therefore the result is either true or false).

Traditional fixed-point computations divide the state space into the states where the fixed-point computation evaluates to *true* and to the states where it evaluates to *false*. However, when computing fixed points in bounded, partial state spaces, this kind of partitioning is usually unachievable. Let us demonstrate this with

Table 1. Truth tables of the three-valued logic

x	$\neg x$
$\perp$	$\perp$
F	T
T	F

$\wedge$	$\perp$	F	T
$\perp$	$\perp$	F	$\perp$
F	F	F	F
T	$\perp$	F	T

$\vee$	$\perp$	F	T
$\perp$	$\perp$	$\perp$	T
F	$\perp$	F	T
T	T	T	T

a simple reachability property $\mathsf{EF}\ p$. When evaluating this property on a bounded state space, we can identify those states where the property evaluates to *true*. From these states a p state is reachable. However, we cannot conclude anything certain about those states from which p is not reachable in this bounded, thus partial state space. Further states must be explored to prove either the *true* or the *false* result. So evaluating $\mathsf{EF}\ p$ on a bounded state space will result $\{true, \perp\}$ unless a sufficiently large state space is explored.

The main advantage of using three-valued logic is that the value *unknown* result suggests that the state space needs to be explored further, and *false* suggests that the model checking procedure is completed. In the classical two-valued logic bounded model checking the *false* result does not provide any information about what caused the property to evaluate to *false*.

As we stated earlier, evaluating a CTL formula in three-valued model checking will label the set of states with one of three possible values: $\{true, false, \perp\}$. Therefore the CTL formula can be described with a three-valued $c : S \to \{true, false, \perp\}$ function. Similarly to the two-valued case, S_{true} represents the set of states where all elements of S are labelled with *true*, $S_{true} = \{s : s \text{ is labelled with } true, s \in S\}$. We define the sets $S_\perp$ and S_{false} similarly.

In the following we introduce the semantics of the basic CTL operators. The semantics of the remaining operators can be derived by duality rules (see Section 2.4).

EX: The traditional EX operator makes a backward step, so the $\mathsf{EX}\ p$ expression evaluates to the $S_{\mathsf{EX}} = \mathcal{N}^{-1}(p)$ two-valued set. This can be generalized with the help of three-valued logic. In the three-valued result set, every $s \in S$ state will be labelled with (where S is the set of all known states):
- *true*, if $s \in S_{\mathsf{EX}}$;
- *false*, if $s \notin S_{\mathsf{EX}}$ and s is not on the border of the bounded state space (e.g there is at least one explored state that is reachable from s);
- $\perp$ otherwise.

EU: The traditional EU operator computes a least fixed point. If S_{EU} is the state set satisfying $\mathsf{E}[p\ \mathsf{U}\ q]$ using two-valued logic, then by using three-valued logic, every $s \in S$ state will be labelled with:
- *true*, if $s \in S_{\mathsf{EU}}$;
- $\perp$, if $s \notin S_{\mathsf{EU}}$ but there exists at least one path from s to the border of the explored (sub)state space whose states are all labelled with p;
- *false* otherwise.

EF: The traditional, two-valued EF operator computes a least fixed point, namely the reachability relation. If S_{EF} is the state set satisfying $\mathsf{EF}\ p$, by using the three-valued EF operator every $s \in S$ state will be labelled with:
- *true*, if $s \in S_{\mathsf{EF}}$;
- $\perp$, if $s \notin S_{\mathsf{EF}}$.

Note that this expression can be evaluated to *false* only if the whole state space is explored.

EG: The traditional EG operator computes a greatest fixed point. If S_{EG} is the state set satisfying $\mathsf{EG}\ p$ using two-valued logic, then by using three-valued logic every $s \in S$ state will be labelled with:
- *true*, if $s \in S_{\mathsf{EG}}$;
- $\perp$, if $s \notin S_{\mathsf{EG}}$, but at least one path exists from s to the border of the state space labelled with p;
- *false* otherwise.

For these interpretations of the EG and EU operators we have to determine whether at least one path exists with p states from a given s state to the border of the explored bounded state space. This computation can be done with the evaluation of another CTL expression. Let S_b represent the state set at the border of the bounded state space representation, i.e., the states having exactly b distance from the initial states. If the expression $\mathsf{E}[p\ \mathsf{U}\ (S_b \cap p)]$ is true for a state set S', then $\forall s \in S'$ there is a path from s to the border (of the bounded state space) containing only p-labelled nodes.

With the help of three-valued logic we can further improve the results produced by model checking, because we can determine if the state space is worth further exploring.

The complexity of three-valued model checking has twice the complexity of traditional bounded model checking. This is due to the fact that at first the algorithm must compute the exact model checking problem based on two-valued logic. If a witness or counterexample is found, then the algorithm will stop. Otherwise the three-valued model checking problem needs to be solved using the extended labelling scheme introduced in this section. The result is the extended information gained from three-valued model checking.

5. EVALUATION

Our algorithm is the first to combine bounded saturation-based state space exploration with saturation-based CTL model checking. We have developed an experimental implementation of the algorithm in the C# programming language. 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.

Our main purpose was to examine the efficiency of our algorithm and compare it to the classical algorithms of CTL model checking. We also examined how saturation-based bounded state space traversal can make CTL-based model checking more scalable. We implemented the saturation-based CTL model checking algorithms from [8] and combined them with bounded saturation-based state space generation.

In this section we compare our bounded model checking algorithm to its classical counterpart from [8]. Our experiments agree with former research results:

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 that need the examination of a larger part of the state space.

The models used for our evaluation are well known in the model checking community. The Flexible Manufacturing System (FMS) and the Kanban system are models of production systems [5]. The parameter N refers to the complexity of the model and it influences the number of tokens in it. Their state space scales from 10^{20} to 10^{30} states. Slotted Ring (SR) is a model of the communication protocol [18], 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 [15]. 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 conciser models. Our models are provided in PNML format, and they can be downloaded from our homepage at http://petridotnet.inf.mit.bme.hu/ (accessed 06.04.2012).

Our algorithm can be fine tuned by the user as both the initial bound and the increment distance can be defined. This way the algorithm can be set to be optimal for smaller distances when we expect that it is sufficient to explore a smaller part of the state space. Moreover, it is also possible to choose both the initial bound and the increment distance bigger to find a proof in fewer iterations, when we assume that the property is "deeper". Some knowledge about the expected behaviour of the property can significantly reduce the computational time.

In Table 2 we compare bounded model checking with the classical approach. We have done measurements with both the *Approximative* and *Exact* truncating function. In Table 2 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 the Hanoi model (which has less concurrency) the *Exact* truncating function has a shorter runtime compared to the *Approximative* algorithm. It is a surprising result, as former research ([19]) stated the contrary. Our (not yet proved) explanation is that our cache-based optimized truncating function is responsible for this speedup. For the other models the *Approximative* truncating function performs better. The reason behind bounded model checking performing well for these models is that the given specification property could always be proven in a bound less than 128. The diameter of their state space is usually much longer: it is 4096 in the case of the Tower of Hanoi, 420 steps in the case of the Kanban model, and scales from 320 to 2800 steps in the case of the FMS model.

In Table 3 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 their runtimes. Bounded model checking works well for the first case, because it takes 10 steps to reach a proof. However, with the growing number of steps needed to (dis)prove a property, the runtime is also growing. The classical model checking algorithm, on the contrary, always needs the same time to prove a property.

Table 4 depicts how the decreasing number of main iterations affects the runtime. If the algorithm increases the bound with bigger "increments", the algorithm will find a proof earlier, reducing the overhead caused by the unsuccessful bounded model checking steps.

Table 2. Runtime results of the algorithms

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
SlottedRing-200	$EF(E_2 = 1 \wedge A_2 = 1)$	1.06 s	2.06 s	88.94 s
SlottedRing-300	$EF(E_2 = 1 \wedge A_2 = 1)$	1.42 s	3.17 s	313.05 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 3. Scaling of the runtime with a growing number of necessary steps

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

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

6. RELATED WORK

Saturation-based bounded state space exploration was presented in [19], where simple reachability and deadlock properties were evaluated. The approach in that paper does not support full CTL model checking. The algorithm served as the base of our bounded model checking algorithm. Saturation-based CTL model checking was presented in [8], where the reader can find a more detailed introduction to saturation-based CTL model checking. Bounded model checking is a widespread technique in the field of hardware verification, for further details see e.g. [1,2,13].

SAT-based bounded approaches proved their efficiency in hardware verification, but there are also efficient techniques for the analysis of Petri nets. We refer the reader to [11,14].

7. CONCLUSION AND FUTURE WORK

We have presented a combined bounded model checking approach for the analysis of Petri nets. Our work is incremental in the sense that we have further improved and combined former approaches and algorithms to get an efficient model checking solution. This paper introduces our first results, and they suggest that there is still much work to be done.

In the near future we would like to make the algorithm capable of building the state space incrementally in each iteration. The main problem now is that truncating the state space invalidates many of the used caches and data structures. For this reason we start building the state space representation from scratch in each new iteration. However, at the expense of losing some distance information, we could utilize the benefits of incremental state space construction by applying a simple transformation. We hope that it would significantly reduce the runtime for large, complex models. We also plan to use the so-called constrained saturation to make the CTL model checking more efficient. Although much work is ahead, hopefully the results given in this paper convinced the reader of the usefulness of our algorithm, and that we successfully combined the efficiency of saturation with bounded model checking.

ACKNOWLEDGEMENTS

This work was partially supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in the frame of the R3-COP project. Dániel Darvas was partially supported by the MFB Hungarian Development Bank Plc. The authors would like to thank Prof. Gianfranco Ciardo for his valuable advice and suggestions.

REFERENCES

1. Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS'99*. Springer-Verlag, London, UK, 1999, 193–207.
2. Bradley, A. R. SAT-based model checking without unrolling. In *VMCAI'11*. Springer-Verlag, Berlin, Heidelberg, 2011, 70–87.
3. Bruns, G. and Godefroid, P. Model checking partial state spaces with 3-valued temporal logics. In *CAV'99*. Springer-Verlag, London, UK, 1999, 274–287.
4. Buchholz, P., Ciardo, G., Donatelli, S., and Kemper, P. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comput.*, 2000, **12**, 203–222.
5. Ciardo, G., Lüttgen, G., and Siminiceanu, R. 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. Springer-Verlag, 2001, 328–342.
6. Ciardo, G., Marmorstein, R., and Siminiceanu, R. The saturation algorithm for symbolic state-space exploration. *Int. J. Softw. Tools Technol. Transf.*, 2006, **8**(1), 4–25.
7. Ciardo, G. and Miner, A. S. Storage alternatives for large structured state spaces. In *Proceedings of the 9th ICCPE: Modelling Techniques and Tools*. Springer-Verlag, London, UK, 1997, 44–57.
8. Ciardo, G. and Siminiceanu, R. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification (CAV'03)*. LNCS 2725. Springer-Verlag, 2003, 40–53.
9. Clarke, E., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, 1999.

10. Clarke, E., McMillan, K., Campos, S., and Hartonas-Garmhausen, V. Symbolic model checking. In *Computer Aided Verification* (Alur, R. and Henzinger, T., eds). Lecture Notes in Computer Science, Vol. 102. Springer, Berlin, Heidelberg, 1996, 419–422.
11. Heljanko, K. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory, CONCUR '01*. Springer-Verlag, London, UK, 2001, 218–232.
12. Kleene, S. *Introduction to Metamathematics*. Bibliotheca mathematica. Wolters-Noordhoff Pub., 1971.
13. McMillan, K. L. Interpolation and SAT-based model checking. In *CAV*, 2003, 1–13.
14. Ogata, S., Tsuchiya, T., and Kikuno, T. SAT-based verification of safe Petri nets. In *ATVA'04*. LNCS 299. Springer, 2004, 79–92.
15. Saad, R., Dal Zilio, S., and Berthomieu, B. Mixed shared-distributed hash tables approaches for parallel state space construction. In *10th International Symposium on Parallel and Distributed Computing (ISPDC 2011), Cluj-Napoca, Romania, July 2011*. IEEE Computer Society. 2011, 9–16.
16. Schuele, T. and Schneider, K. Three-valued logic in bounded model checking. In *MEMOCODE'05*. IEEE Computer Society, Washington, DC, USA, 2005, 177–186.
17. Skolem, T. A set theory based on a certain 3-valued logic. *Mathematica Scandinavica*, 1960, **8**, 127–136.
18. Vörös, A., Bartha, T., Darvas, D., Szabó, T., Jámbor, A., and Horváth, Á. Parallel saturation based model checking. In *10th International Symposium on Parallel and Distributed Computing (ISPDC 2011), Cluj-Napoca, July 2011*. IEEE Computer Society, 2011, 94–101.
19. Yu, A., Ciardo, G., and Lüttgen, G. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *Int. J. Softw. Tools Technol. Transf.*, 2009, **11**, 117–131.
20. Zhao, Y. and Ciardo, G. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *ATVA'09*. Springer-Verlag, Berlin, Heidelberg, 2009, 368–381.

Tõkestatud küllastamisel põhinev arvutuspuude loogikas (CTL) väljendatud mudelkontroll

András Vörös, Dániel Darvas ja Tamás Bartha

Formaalne verifitseerimine on muutumas ohutuskriitilise ja mudelpõhise tarkvaraarenduse oluliseks osaks. Verifitseerimisprotsessi osana on mudelkontrollitehnika üks enim väljaarendatud viise, kuidas süsteemi käitumist analüüsida. Mudelkontroll sümbolkujul on tõhus lähenemisviis, käsitlemaks isegi keerulisi ja suure olekuruumiga mudeleid. Küllastamine on spetsiaalset iteratsioonistrateegiat kasutav sümbolalgoritm, mis on tõhus asünkroonsete mudelite kontrollimisel. Viimaste aastate edusammude tulemusena on leiutatud mitmeid uusi küllastamisel põhinevaid algoritme, näiteks olekuruumi genereerimiseks, piiratud olekuruumi genereerimiseks, aga ka struktuurseks mudelkontrolliks. Käesolevas artiklis on uuritud, kuidas sobib süsteemi verifitseerimiseks kahe täiustatud mudelkontrolli algoritmi – nimelt piiratud küllastamise ja küllastamisel baseeruva struktuurse mudelkontrolli algoritmi – kombinatsioon. Meie töö on esimene katse ühendada nimetatud kaks lähenemist ja sel viisil käsitleda ning uurida keerulisi või isegi lõpmatu olekuruumiga süsteeme. Meie mõõtmised näitavad, et küllastamine on tõhus ka tõkestatud mudelkontrolli korral.