Efficient Saturation-based Bounded Model Checking of Asynchronous Systems

Dániel Darvas[1], András Vörös[1], and Tamás Bartha[2]

[1] Dept. of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
vori@mit.bme.hu
[2] Computer and Automation Research Institute
MTA SZTAKI,
Budapest, Hungary

Abstract. Formal verification is becoming a fundamental step in assuring the correctness of safety-critical systems. However, due to these systems being often asynchronous and even distributed, their behaviour can be very complex. Thus, their verification necessitates methods that can deal with huge or even infinite state spaces. In this paper we present how the integration of two advanced algorithms for asynchronous systems —namely *bounded saturation* and *constrained saturation-based structural model checking*— can be used to verify such systems.

Model checking is one of the current advanced techniques to analyse the behaviour of systems, as part of the verification process. The so-called *saturation* algorithm has an efficient iteration strategy combined with symbolic data structures, providing a powerful state space generation and model checking solution for asynchronous systems. *Bounded saturation* utilizes the efficiency of saturation in bounded state space exploration. *Constrained saturation* is an efficient structural model checking algorithm. Our work is the first approach to integrate these algorithms. Our measurements confirm that the new approach does not only offer a way dealing with even infinite state spaces, but in many cases even outperforms the original methods.

1 Introduction

Assuring the quality of safety critical, embedded systems is a challenging task. Advances in technology are making it even more difficult: components are becoming more complex, and systems have more components that interact using complicated communication and synchronisation mechanisms. Due to this complexity it is impossible to make claims about the correctness of these systems without the help of *formal methods*. On the other hand, exactly this complexity raised the need for highly efficient formal verification algorithms.

Formal verification usually starts with the creation of a formal model of the studied system. Then the behaviour of the formal model is analysed in order to prove its adequacy. One of the most prevalent analysis techniques is *model checking* [4], an automatic technique to check whether the model (and thus the modelled system) satisfies its specification. The specification is typically expressed in *temporal logic*. *Computation Tree Logic* (CTL) is a popular temporal logic language due to the efficient and relatively simple analysis algorithms supporting it.

Model checking traverses the state space of the model being analysed. Safety critical systems are often asynchronous, even distributed, therefore the composite state space of their asynchronous subsystems can be as large as the Cartesian product of the local components' state spaces, thus the state space of the whole system explodes. *Symbolic methods* [4] are advanced techniques to handle huge state spaces of synchronous systems. 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. Ordinary symbolic methods, however, usually perform poorly for asynchronous systems.

Saturation [1] is considered as one of the most effective state space generation and model checking algorithms for asynchronous systems It combines the efficiency of symbolic methods with a special iteration strategy. Saturation-based state space exploration computes the set of reachable states. The so-called *saturation-based structural model checking* algorithm can analyse temporal logic properties. Nowadays, the so-called *constrained saturation-based structural model checking* algorithm is one of the most efficient algorithms for model checking [11].

Nevertheless, there are still many complex models that have a state space, which is either too large to be represented even symbolically, or it is infinite. In these cases *bounded model checking* can be a solution, as it explores a bounded part of the state space, and examines the prescribed properties on it. *Bounded saturation-based state space exploration* was presented in [10], where the authors introduced a new saturation algorithm that explores the state space only to some bounded depth.

1.1 Motivation

Former approaches solved only one of the problems: they could either be used for structural model checking over the entire state space; or they could traverse the state space up to a given bound, but without being able to check complex properties on it. In this paper we introduce a new saturation-based bounded model checking algorithm that integrates both approaches. Our algorithm incrementally explores the state space and performs structural model checking on the uncovered bounded part. To our best knowledge, this is the first attempt to combine bounded saturation-based state space exploration with constrained saturation-based CTL model checking, in order to gain the advantages of both techniques.

Furthermore, bounded model checkers usually do not support full CTL. Even though there were theoretical results in this area, former bounded model checking approaches did not work well with CTL due to its branching characteristics.

Our work is a step towards efficient bounded CTL model checking with many directions to be explored in the future.

The structure of our paper is as follows: section 2 introduces the background and prerequisites of our work. Section 3 gives an overview of the advanced saturation-based algorithms our work relies on. Section 4 describes the new bounded CTL model checking algorithm and its details. Section 5 presents our measurements results. At the end our conclusions and ideas for future work complete the paper.

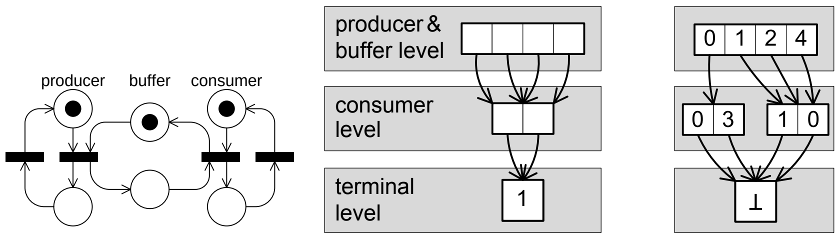
2 Background

In this section we outline the theoretical background of our work. First, we describe the underlying data structures of our algorithms used for storing the state space during model checking: *Multiple-valued Decision Diagrams* (MDDs) and *Edge-valued Decision Diagrams* (EDDs). EDDs extend MDDs with extra information: in addition to storing the state space they also provide the distance information for bounded state space generation. Finally, we summarize the saturation-based state space exploration algorithm and the model checking background.

2.1 Decision Diagrams

This section is based on [9]. Decision diagrams are used in symbolic model checking for efficiently storing the state space and the possible state changes of the models. 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 (terminal 0 and terminal 1). The nodes are ordered into $K + 1$ levels. A non-terminal node is labelled by a variable index $1 \leq k \leq K$, 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. These rules ensure that MDDs 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(a) depicts a simple example Petri net [7] 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 containing 8 states. 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 [1], and the possible (global) states are the paths from the root node to the terminal *one* node. (The model has to be decomposed to be able to represent its state space using decision diagrams efficiently. This decomposition will be discussed in Section 2.3.)



(a) The Petri net of producer-consumer model

(b) State space representation with MDD

(c) State space and state distance representation with EDD

Fig. 1. Producer-consumer example

An *Edge-valued Decision Diagram* (EDD) is an extended MDD that can represent the following function: $f : \{0, 1, \ldots\}^K \to \mathbb{N} \cup \{\infty\}$. Figure 1(c) depicts an EDD storing the encoded state space enriched with the distance information (computed from the initial state). The differences between an MDD and an EDD are the following:

- Every p node is visualized as a rectangle with k slots, where k is the number of children (domain of the variable).
- 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. 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. 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 which goes to the terminal zero node. Usually the zero valued dangling edges and the ∞ valued edges are not shown.
- Every non-terminal node has an outgoing edge with weight 0.

In the example of Figure 1(c) 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 $x[1] = \langle\perp, 3\rangle$.

2.2 Model Checking and Bounded Model Checking

Given a formal model, *model checking* [4] is an automatic technique to decide whether it satisfies the specification. Formally: let M be a Kripke structure (i.e., a labelled 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 s.t. $M, s \vDash f$.

Bounded model checking decides whether the model satisfies the specification in a predefined number of steps, which is the depth of the state space traversal. Formally: let M be a Kripke structure. Let f be a formula of temporal logic. The bounded model checking problem for the k bounded state space is to find all states s of M such that $M, s \vDash_k f$. Among other cases, bounded model checking is useful if the full state space is not needed to decide on a property. This is e.g. the case for *shallow bugs* that can be found with bounded model checking efficiently.

Structural model checking uses a set operations to evaluate temporal logic specifications by computing fixed-points in the state space. *CTL* (Computation Tree Logic) [4] is widely used temporal logic specifications formalism, as it has expressive syntax, and structural model checking yields efficient algorithms to analyse CTL specifications. CTL expressions contain state variables, Boolean operators, and *temporal operators*. Temporal 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, only three: EX, EU, EG of the 8 possible pairings need to be implemented due to duality [4]. The remaining five can be expressed with the help of the former three in the following way: $\mathsf{AX}p \equiv \neg\mathsf{EX}\neg p$, $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$, $\mathsf{AF}p \equiv \neg\mathsf{EG}\neg p$, $\mathsf{A}[p\mathsf{U}q] \equiv \neg\mathsf{E}[\neg q\ \mathsf{U}(\neg p \wedge \neg q)] \wedge \neg\mathsf{EG}\neg q$, $\mathsf{EF}p \equiv \mathsf{E}[\mathit{true}\ \mathsf{U}\ p]$.

2.3 Saturation

Saturation is a *symbolic algorithm* for state space generation and model checking. *Decomposition* serves as the prerequisite for the symbolic encoding: the algorithm maps the state variables of the chosen high-level formalism into symbolic variables of the decision diagram. The global state of the model can be represented as the composition of the local states of components: $s_g = (s_1, s_2, \ldots, s_n)$, where n is the number of components. See Figure 1(b) for a possible decomposition and the corresponding MDD representation of the example model in Figure 1(a). Furthermore, decomposition helps the algorithm to efficiently exploit *locality*, which is inherent in asynchronous systems. Locality ensures that a transition usually affects only some components or some parts of the submodels. The algorithm does not create a large, monolithic next state function representation. Instead it divides the global next state function $\mathcal{N}$ into smaller parts, according to the high-level model. Formally: $\mathcal{N} = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e$, where $\mathcal{E}$ is the set of events in the high level model. The granularity of the decomposition, i.e. the next state relations represented by $\mathcal{N}_e$ can be chosen arbitrarily [3].

Saturation uses *symbolic encoding of the next state function.* In our work we use the symbolic next state representation from [3]. This approach partitions disjunctively the global next state function according to the high level model events in the system: $\mathcal{N} = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e$. Logically, if $\mathcal{N}$ is represented by the relation between state variables (in the decision diagram representation) $\boldsymbol{x}, \boldsymbol{x}'$ with $\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}')$, then the global relation can be expressed by the symbolic next state relations of the events: $\mathcal{R}(\boldsymbol{x}, \boldsymbol{x}') = \bigvee_{e \in \mathcal{E}} \mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}')$. This way the algorithm can use smaller next state representations. However, in many cases the computation of the local $\mathcal{N}_e$ functions is still expensive. The algorithm handles this problem by conjunctive partitioning according to the enabling and updating functions (denoted by $\mathcal{N}^{enable}$ and $\mathcal{N}^{update}$) [3]: $\mathcal{N}_e = \bigcap_{\forall i}(\mathcal{N}_{e,i}^{enable} \bigcap \mathcal{N}_{e,i}^{update})$, which can be symbolically computed by the following equation: $\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}') = \bigwedge_{\forall i}(\mathcal{R}_{e,i}^{enable}(\boldsymbol{x}, \boldsymbol{x}') \bigwedge \mathcal{R}_{e,i}^{update}(\boldsymbol{x}, \boldsymbol{x}'))$. Applying $\mathcal{N}_e$ to a given set of states represented by *states* results in $\mathcal{N}_e(\mathit{states}) = \mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}'), \mathit{states})$, where *RelProd* is the well-known relational product function [3]. The smaller the partitions we create, the less computation they need. The limit for the size of the partitioning comes from the used high level modelling formalism.

Saturation uses a *special iteration strategy*, which is efficient for asynchronous systems. The construction of the MDD representation of the state space starts by building the MDD representing the initial state. Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively, if new states are discovered. 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. The result is the state space representation encoded by MDD.

Saturation-based Structural Model Checking. Saturation-based structural CTL model checking was first presented in [2], where the authors introduced how the least fixed point operators can be computed with the help of saturation. CTL model checking explores the state space in a backward manner. It constructs the inverse representation $\mathcal{N}^{-1}$ and computes the inverse next state, greatest and least fixed points of the operators. The semantics of the three implemented CTL operators [4] is:

- **EX**: $i^0 \vDash \mathsf{EX}\ p$ iff $\exists i^1 \in \mathcal{N}(i^0)$ s.t. $i^1 \vDash p$. This means that EX corresponds to the function $\mathcal{N}^{-1}$, applying one step backward through the next state relation.
- **EG**: $i^0 \vDash \mathsf{EG}\ p$ iff $i^0 \vDash p$ and $\forall n > 0, \exists i^n \in \mathcal{N}(i^{n-1})$ 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 fixed point, however, benefits from the locality accompanying the decomposition.
- **EU**: $i^0 \vDash \mathsf{E}[p\ \mathsf{U}\ q]$ iff $i^0 \vDash p$ and $\exists n > 0, \exists i^1 \in \mathcal{N}(i^0), \ldots, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash q$ and $i^m \vDash p$ for all $m < n$ (or $i^0 \vDash q$). The states satisfying this property are computed with the following least fixed-point: $\mathbf{lfp}\ Z[q \vee (p \wedge \mathsf{EX}\, Z)]$ Informally: we search for a state q reached through only states satisfying p.

3 Bounded and Constrained Saturation

In this section we give an overview of the two saturation-based advanced algorithms that form important parts of our new approach. *Bounded saturation* is used for state space exploration. *Constrained saturation* is used to restrict structural model checking to the bounded state space. The integration of constrained saturation with the bounded saturation-based state space generation lead to the first saturation-based bounded model checking algorithm, which exploits the efficiency of structural model checking for bounded state spaces.

3.1 Bounded Saturation

It is difficult to exploit the efficiency of saturation for bounded state space exploration, because saturation uses an irregular recursive iteration order, which is totally different from traditional breadth-first traversal. Consequently, bounding the recursive exploration steps of saturation does not necessarily guarantee this bound to be global for the state space representation.

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 that has already proved its efficiency [10]. Although MDDs provide a highly compact solution for state space representation, bounded saturation needs additional distance information during the traversal. For this reason, [10] uses Edge-valued Decision Diagrams (EDDs) instead of MDDs, and —in addition to the state space— it also encodes the minimal distance of each state from the initial state(s) into the EDD. The algorithm first iterates through the state space until a given bound is reached, which is represented by an edge in the EDD. After that it cuts the parts that are beyond the depth of the traversal from the EDD, thereby computing the reachability set below the bound.

In our previous work [9] we extended the algorithm [10] with on-the-fly updates [1] and an additional caching mechanism.

3.2 Constrained Saturation

In [11] the authors introduced an advanced saturation-based iteration strategy for the purpose of structural model checking. The algorithm, called *constrained saturation*, computes the least fixed point of the reachability relation that satisfies a given constraint.

The main novelty of the new algorithm is the slightly different iteration style. Instead of combining saturation with breadth-first traversal, it uses a pre-checking phase. The algorithm builds on the following observation [11]: in order to do the symbolic step $\mathcal{N}_e$ from the set of state *states* to a set of states satisfying the constraint $\mathcal{C}$, we have to compute $\mathcal{N}_e(\mathit{states}) \cap \mathcal{C}$. This contains an expensive intersection operation after each step. Using the following observation: $\mathcal{N}_e(\mathit{states}) \cap \mathcal{C} = \mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}'), \mathit{states}) \cap \mathcal{C} = \mathit{RelProd}(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}') \wedge \boldsymbol{x}' \cap \mathcal{C} \neq 0, \mathit{states})$ the algorithm can use pre-checking phase and it avoids the computation-intensive intersection operation after the symbolic state space step, instead it simply skips those steps which would go out of the constraint [11].

Algorithms 1 and 2 formalize the operation of the constrained saturation algorithm. The lines starting with $*$ are the additions to traditional saturation. In Algorithm 1 it is easy to see that the *ConsSaturate*(c,s) computes

Algorithm 1: ConsSaturate

```
input   : c, s : node
// c: constraint,
// s: node to be saturated
output : node

  l ← s.level; r ← N_l^{-1};
  t ← NewNode(l);
  foreach i ∈ S_l : s[i] ≠ 0 do
*   if c[i] ≠ 0 then
      t[i] ← ConsSaturate(c[i], s[i]);
    else
*     t[i] ← s[i];
  repeat
    foreach i, i' ∈ S_l : r[i][i'] ≠ 0 do
*     if c[i'] ≠ 0 then
        u ← RelProd(c[i'], t[i], r[i][i']);
        t[i'] ← Union(t[i'], u);
  until t unchanged;
  t ← CheckIn(l, t);
  return t;
```

Algorithm 2: RelProd

```
input   : c, s, r : node
// c: constraint,
// s: node to be saturated,
// r: next state function
output : node

  if s = 1 ∧ r = 1 then return 1;
  ;
  l ← s.level; t ← 0;
  foreach i, i' ∈ S_l : r[i][i'] ≠ 0 do
*   if c[i'] ≠ 0 then
      u ← RelProd(c[i'], t[i], r[i][i']);
      if u ≠ 0 then
        if t = 0 then
          t ← NewNode(l);
        ;
        t[i'] ← Union(t[i'], u);
  t ← CheckIn(l, t);
  t ← ConsSaturate(c, t);
  return t;
```

$RelProd(\mathcal{R}_e(\boldsymbol{x}, \boldsymbol{x}') \cap \mathcal{C}, states)$ without using the expensive symbolic intersection operation. Research showed [11] that *ConsSaturate* is faster than traditional saturation when there is a constraint on the possible states. This is the situation e.g. in the case of the EU CTL operator.

4 Efficient Saturation-based Bounded Model Checking

In this section we present our new, saturation-based bounded model checking algorithm. In order to have an efficient model checking procedure that produces the model checking result from the specification and the formal model, the following ingredients are needed:

- an efficient state space exploration method,
- an efficient model checking algorithm,
- a powerful search strategy,
- a mechanism to decide on the specification.

We use bounded saturation to efficiently explore the bounded state space and produce a symbolic representation. After that we employ constrained saturation-based model checking to provide full CTL model checking on this state space.

4.1 Constrained Saturation using the Bounded State Space

Many model checking tools limit the specification syntax to a subset of the CTL temporal language, in order to simplify the analysis task and boost performance.

We want to support the full CTL semantics in model checking, and thus we must use backward traversal. This is our main reason for choosing the traditional, fixed-point–based algorithms; as the semantics of forward and backward CTL model checking are different (and incomparable) [5].

The naive approach to combine bounded exploration and structural model checking would be to apply the fixed point computations from the bounded state space on the complete lattice. However, the efficiency of this naive approach would converge to traditional fixed point computations. It could be improved by constructing the intersection of the result from the fixed point iterations with the bounded state space representation, practically restricting each iteration of the fixed point computation to the bounded subspace. All the same, the improvement still suffers from poor performance due to the extensive use of the costly intersection operation.

Our aim is to utilize the saturation approach also during model checking, and exploit the constrained saturation iteration strategy to provide an efficient bounded model checking algorithm supporting the full CTL semantics. The main idea is that the symbolically encoded explored bounded state space can serve as the constraint in the constrained saturation algorithm. This way we can expeditiously bound the least fixed point computations. Below we define how the constrained saturation decides on the following CTL operators (where **lfp** denotes the least fixed-point, and *bss* denotes the bounded state space as represented by the MDD):

- **EF**: $M, s \vDash_k \mathsf{EF}p$ iff $s_0 \subseteq \mathbf{lfp}\ Z[(p \wedge bss) \vee (bss \wedge \mathsf{EX}\, Z)] = \mathit{ConsSaturation}(bss, p \cap bss)$. This way we can directly exploit the constrained saturation algorithm to produce the least fixed point in the given bounded state space *bss*. The result can be utilised by other, both least and greatest fixed point operators.
- **EU**: $M, s \vDash_k \mathsf{E}[p \mathsf{U} q]$ iff $s_0 \subseteq \mathbf{lfp}\ Z[(q \wedge bss) \vee (bss \wedge p \wedge \mathsf{EX}\, Z)] = \mathit{ConsSaturation}(bss \cap q, bss \cap p)$. This is similar to using the constrained saturation algorithm in traditional saturation-based model checking [11], but within a bounded setting. This result can also be nested into both least and greatest fixed point operators.

As greatest fixed point computations (EG) and simple next state operators (EX) does not require such restrictions in the exploration, we apply traditional fixed point algorithms for them. Although operator EF is just a special case of operator EU, for performance reasons it is worth to be implemented separately.

4.2 Search Strategies

The choice of the search strategy followed during bounded model checking has a significant impact on performance. In this section we evaluate the possible search strategy alternatives. With regard to bounded state space generation, we can have two approaches:

- Given a fixed bound b, we explore the b bounded state space and evaluate the specification on it. We call it the *fixed bound strategy*.

– Given an initial bound *init* and increment value *inc*, we start exploring the state space to the given bound *init*. The model checking algorithm then decides whether it can stop, or it has to increase the bound by *inc*. The procedure stops when it runs out of resources, or the model checking question is answered. We call it the *incremental strategy*.

Traditional bounded model checking uses the increasing depth incremental strategy, typically looking one step further in the state space in a breadth-first manner. Applying this strategy in saturation would lead to lose the efficiency of the special iteration order of saturation. Our experience shows that it is better to let saturation increase the depth by at least 5–10 steps. Finding a good trade-off in choosing the iteration depth is important. A one-step iteration results in the loss of efficiency during saturation. On the other hand, a too large increase of iteration depth results in the loss of efficiency during bounded model checking. We have developed two different incremental search strategies:

– The *restarting strategy* starts again the iteration from the initial state after each iteration, and uses the increased bound in the exploration.
– The *continuing strategy* reuses the formerly explored bounded state space as the set of initial states in the next iteration, and extends it using the bounded saturation algorithm to represent the state space of the increased bound.

The restarting strategy was straightforward to implement, since it simply uses the bounded saturation algorithm. For the continuing strategy we had to modify the bottom-up building strategy of the saturation algorithm. For this purpose, we needed to extend the algorithm to be able to handle even huge initial state sets. This extension contained the modification of the *truncating operations*, the *caching mechanisms* in order to preserve correctness, and the *construction of the decision diagram representation* to be able to handle huge initial set of states. The continuing strategy uses the formerly built data structures which can be more efficient than building every data structure from scratch at each iteration.

4.3 Decision Mechanism

It is also important to be able to decide if the specification is satisfied. Bounded model checking is a semi-decision procedure, therefore it can be used to ensure the following behavioural properties of the specification:

– *Invariant and safety*: proving these properties needs the full state space to be explored, or bounded model checking can give a short counterexample (witness), if it exists.
– *Liveness*: bounded model checking can find a short witness to these properties, or the full state space has to be explored to refute them.
– Other properties, such as combination of safety and liveness properties: 3-valued logic can be used for decision.

Invariant and safety properties are usually proved (in symbolic model checking) by finding inductive invariants without exploring the full state space. This approach cannot be used directly for liveness properties.

Finding Inductive Proof against Liveness Properties. EDD-based state space representation helps us to tell more about liveness properties. Refuting liveness properties may come from the fact that: (1) the algorithm has to explore more from the state space to find a witness, (2) the liveness property does not hold, and there exists a counterexample in the bounded state space. Our approach can handle these differences. This is in contrast to traditional bounded model checking approaches, since they have to encode the difference of the two cases into the SAT formula directly, which is inefficient.

If a liveness property $\mathsf{EG}\, p$ does not hold in the bounded state space *bss*, we can decide whether to investigate the state space further, or to conclude that it will never hold. Let $p_{d=bound}$ be the set of states, where p is true and their distance from the initial state is $d = bound$. $p_{d=bound}$ is encoded in the EDD, we need to traverse the EDD once to get this state set. It can be computed efficiently from the symbolic encoding. Let $result = \mathbf{lfp}\ Z[p_{d=bound} \vee (p \wedge \mathsf{EX}\, Z)] = ConsSaturate(p, p_{d=bound})$, then $s_0 \wedge result = false \Rightarrow \mathsf{EG}\, p = false$ holds.

4.4 Summary of Our Contributions

In this section we described the first efficient saturation-based bounded model checking algorithm, which combines the efficiency of constrained saturation and bounded state space exploration. It has the following properties:

- $\forall f(Z)$: $\mathbf{fp}\ f(Z) \subseteq bss$, for all fixed point the bounded saturation algorithm is bounded by the state space, even for the least fixed point computations.
- It is efficient from the model checking point of view as the algorithm traverses the bounded state space with the saturation iteration strategy.
- With the creative use of constrained saturation it avoids to examine states outside of the discovered bounded state space in the model checking phase.
- It avoids expensive intersection operators during the state traversal of least fixed point operators.

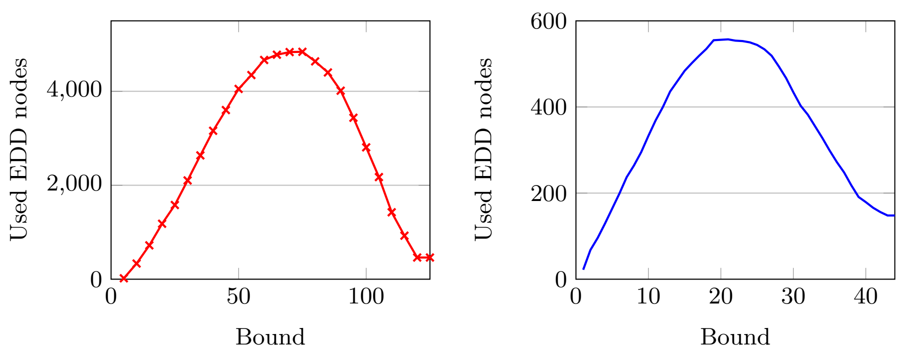
5 Evaluation

We have performed measurements in order to confirm that the presented novel constrained saturation-based bounded model checking algorithm performs better than former approaches. This section summarizes our measurement results.

Our aim was to examine the efficiency of our new algorithm and compare it to a classical saturation-based structural model checking algorithm. We have also examined how saturation-based bounded state space traversal can make CTL-based model checking more scalable. For this purpose we have developed an experimental implementation of our algorithm using the C# programming language. We have also implemented the algorithm taken from [11] as the reference for comparison, which we denoted in the measurements as “Unbounded”. For the measurements we used a desktop PC (Intel Q8400 2.66 GHz CPU, 4 GB memory with Windows 7 x64 and .NET 4.0 framework).

The models we used for the evaluation are widely known in the model checking community. We took the models of *Tower of Hanoi* from [9]. The state space of the Tower of Hanoi models scales from 531 441 up to $3,5 \cdot 10^9$ states. The saturation algorithm does not perform well for this model, as it does not correspond to an asynchronous system. These measurements demonstrate that our bounded model checking algorithm can analyse even those models, which are not well suited for saturation. The *Slotted Ring* (SR) is the model of a communication protocol [1], [8]. The size of the state space of the SR–100 model is about 10^{100} states. The *Flexible Manufacturing System* (FMS–N) is a model of production systems [1]. The parameter N refers to the complexity of the model checking problem. For $N = 20$ the state space of the FMS model has 10^{20} states.

Both the initial bound and the increment distance are changeable parameters, thus our algorithm can be fine tuned by the user. If the properties to prove are expected to be "shallow", then the algorithm can be set to work optimally for smaller distances. On the other hand, when the properties to prove are "deeper", then both the initial bound and the increment distance can be set bigger to find a proof in fewer iterations. A priori knowledge about the expected behaviour of the properties can significantly reduce the computational time.



(a) Slotted Ring (SR–10)

(b) FMS–3

Fig. 2. Size of state space representation (EDD) at each iteration

Table 1 lists our run time measurements for simple reachability properties of the structural model checking (Unbounded), and our bounded model checking approach (Bounded, incremental, restarting strategy). Saturation-based model checking is extremely efficient for asynchronous systems, and the modified iteration strategy requires more computational resources, so one would expect that for these models the traditional approach is better. In the case of Slotted Ring (SR–N, where N is the number of components) models, the analysed property was the following: $\mathsf{E}(B_1 \neq 1 \vee F_1 \neq 1 \;\mathsf{U}\; G_2 = 1 \wedge A_2 = 1)$. The advantage of

Table 1. Comparing run times of model checking for different asynchronous models

Model	**Unbounded**	**Bounded, incremental, restarting strategy**
SR–100	$>$ 1800 s	15.99 s
SR–200	$>$ 1800 s	38.12 s
SR–300	$>$ 1800 s	49.82 s
RR–100	0.24 s	0.27 s
RR–200	0.47 s	0.05 s
RR–1000	2.61 s	0.28 s
RR–10 000	32.54 s	3.39 s
DPhil–10	0.05 s	0.04 s
DPhil–100	0.40 s	0.53 s
DPhil–1000	5.26 s	5.14 s
DPhil–3000	16.19 s	19.52 s
DPhil–10 000	79.64 s	323.26 s

Table 2. Tower of Hanoi model checking run time results

Model	**Unbounded**	**Bounded, incremental, restarting strategy**	**Bounded, incremental, continuing strategy**	**Bounded, fixed bound**
Hanoi–12	39.2 s	6.45 s	2.15 s	1.62 s
Hanoi–14	$>$ 1800 s	6.85 s	2.38 s	1.76 s
Hanoi–16	$>$ 1800 s	10.09 s	2.72 s	1.92 s
Hanoi–18	$>$ 1800 s	10.80 s	3.09 s	2.04 s
Hanoi–20	$>$ 1800 s	11.26 s	3.12 s	2.64 s

Table 3. Comparing strategies for complex properties

Model	**Unbounded**	**Bounded, incremental, restarting strategy**	**Bounded, incremental, continuing strategy**	**Bounded, fixed bound**
FMS–25	1.70 s	1.01 s	1.14 s	0.39 s
FMS–50	9.58 s	2.37 s	3.00 s	1.03 s
FMS–100	82.39 s	4.88 s	6.55 s	1.93 s
FMS–1000	$>$ 1800 s	5.58 s	6.49 s	1.93 s
FMS–10 000	$>$ 1800 s	5.60 s	7.16 s	1.91 s
FMS–1 000 000	$>$ 1800 s	5.68 s	7.11 s	1.95 s

bounded model checking is revealed by the model, as traditional model checking runs out of resources even for such a simple property.

We have also examined Round-Robin models (RR–N, where N is the number of components), which are quite efficiently handled by the traditional saturation based model checking approach. We chose the following property to be checked: $\mathsf{E}(pload_1 = 0 \;\mathsf{U}\; psend_0 = 1)$. This property is shallow, so the advantage of our bounded model checking approach is well reflected in the results.

The model of the Dining Philosophers (DPhil–N, where N is the number of philosophers) revealed that for those models, where the saturation algorithm answers the model checking question (in this case: $\mathsf{E}(\neg eating_2 \;\mathsf{U}\; eating_1)$) extremely fast, bounded model checking is slower. The reason for this is that the overhead of bounded model checking simply does not pay off.

In Table 2 and Table 3 we compare the different approaches for complex properties. Table 2 contains the measurements of the Tower of Hanoi models. We have examined a combined safety-liveness property ($\mathsf{EG}(\mathsf{EF}(B_{\downarrow 8} > 0))$, where $B_{\downarrow 8} > 0$ denotes the placement of the 8th disk to the 2nd rod). The traditional structural model checking approach (Unbounded) runs out of resources early. Knowing the exact bound can help the algorithm to answer the model checking question as fast as possible (Bounded, fixed bound). Comparing the two different bounded model checking strategies, the continuing strategy has advantage as it uses up the formerly computed results during the model checking.

In Table 3 the run time results for the property $\mathsf{EG}(\mathsf{E}(M1 > 0 \;\mathsf{U}\; (P1s = P2s = P3s = 8)))$ of the model FMS are depicted. This property is also a combined safety-liveness property that represents the existence of a circle in a certain set of states satisfying some safety requirements (based on [2]). The structural model checking algorithm time-outs for big parameters. By setting an adequate bound, the bounded model checking approach answers the model checking question very fast (Bounded, fixed bound). When we compare the two bounded model checking strategies, the result is surprising: the restarting strategy solves the model checking problem for every parameter *faster* than the continuing strategy. We investigated the reason for this. It can be seen in Figure 2 that for asynchronous systems (like FMS) the state space representation grows steeply up to a given value, but after that it starts decreasing (resembling a bell curve). The continuing strategy uses these intermediate state space representations as the initial state, which is a large computational overhead compared to starting the iteration from the initial state. By beginning model checking from scratch (i.e., using the restarting strategy) we can exploit the efficiency of saturation for building the state space representation. By starting to modify an intermediate representation (i.e., using the continuing strategy) the algorithm has to do more computations, especially if the intermediate representation is larger than the final one.

6 Conclusion and future work

We have presented in this paper an advanced bounded model checking approach based on the saturation algorithm. Our work exploits the efficiency of saturation and enables us to verify complex, or even infinite-state models. Our approach also extends the set of asynchronous systems that can be analysed with the help of symbolic methods. We have proved the efficiency of the new approach with measurements.

We intend to develop our solution further. We will investigate the use of forward model checking [6] instead of the classical backward fixed point computation, as we believe this can further improve the performance of our algorithm. We also plan to use the constrained saturation algorithm in a different way, in order to avoid redundant computations more efficiently.

References

1. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 379–393. Springer (2003)
2. Ciardo, G., Siminiceanu, R.: Structural symbolic CTL model checking of asynchronous systems. In: Computer Aided Verification (CAV'03), LNCS 2725. pp. 40–53. Springer-Verlag (2003)
3. Ciardo, G., Yu, A.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. Correct Hardware Design and Verification Methods 3725, 146–161 (2005)
4. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
5. Henzinger, T., Kupferman, O., Qadeer, S.: From pre-historic to post-modern symbolic model checking. In: Computer Aided Verification. pp. 195–206 (1998)
6. Iwashita, H., Nakata, T.: Forward model checking techniques oriented to buggy designs. ICCAD-97 pp. 400–404 (1997)
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
8. Vörös, A., Bartha, T., Darvas, D., Szabó, T., Jámbor, A., Horváth, Á.: Parallel saturation based model checking. In: ISPDC11. IEEE Computer Society (2011)
9. Vörös, A., Darvas, D., Bartha, T.: Bounded Saturation Based CTL Model Checking. In: Penjam, J. (ed.) Proc. of the 12th Symposium on Programming Languages and Software Tools, SPLST'11. pp. 149–160. Tallinn, Estonia (2011)
10. Yu, A., Ciardo, G., Lüttgen, G.: Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. Int. J. Softw. Tools Technol. Transf. 11, 117–131 (2009)
11. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis. pp. 368–381. ATVA '09, Springer-Verlag, Berlin, Heidelberg (2009)

Acknowledgement

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