Acta Cybernetica **22** (2016) 573–589.

Improving Saturation-based Bounded Model Checking*

Dániel Darvas†, András Vörös†, and Tamás Bartha‡

Abstract

Formal verification is becoming a fundamental step in assuring the correctness of safety-critical systems. Since these systems are often asynchronous and even distributed, their verification requires methods that can deal with huge or even infinite state spaces. Model checking is one of the current techniques to analyse the behaviour of systems, as part of the verification process.

In this paper a symbolic bounded model checking algorithm is presented that relies on efficient saturation-based methods. The previous approaches are extended with new bounded state space exploration strategies. In addition, constrained saturation is also introduced to improve the efficiency of bounded model checking. Our measurements confirm that these approaches do not only offer a solution to deal with infinite state spaces, but in many cases they even outperform the original methods.

Keywords: model checking, symbolic model checking, CTL, bounded model checking, saturation, asynchronous systems, compacting saturation

1 Introduction

The usage of formal methods is one of the possible solutions to assure the quality of the more and more complex safety-critical, embedded systems. For this reason, highly efficient formal verification algorithms are needed. *Model checking* is one of the most prevalent formal verification technique. It is an automatic approach to check whether the formal model (and thus the modelled system) satisfies its specification. This specification can be expressed e.g. in *Computation Tree Logic* (CTL) that is a popular temporal logic language thanks to the efficient verification algorithms.

*This work was partially supported by the ARTEMIS JU and the Hungarian Ministry of National Development (NFM) in frame of the R5-COP (Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems) project.

†Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary. E-mail: darvas@mit.bme.hu, vori@mit.bme.hu.

‡Institute for Computer Science and Control, MTA SZTAKI, Budapest, Hungary.

DOI: 10.14232/actacyb.22.3.2016.2

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, i.e., the state space of the whole system "explodes". *Symbolic methods* [7] 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. Ordinary symbolic methods, however, usually perform poorly for asynchronous systems.

Saturation [3] 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 evaluate temporal logic properties. Nowadays, the so-called *constrained saturation-based structural model checking* algorithm is one of the most efficient algorithms for CTL model checking [13].

However, the state space of many complex models is either infinite, or too large to be represented even symbolically. In these cases *bounded model checking* can be a solution, as it explores and examines the prescribed properties on a bounded part of the state space. *Bounded saturation-based state space exploration* was introduced in [12], i.e., an algorithm that explores the state space only to some bounded depth.

Motivation and Previous Work. Former saturation-based 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 checking complex properties. In the method presented here these two approaches are integrated. Our algorithm incrementally explores the state space and performs structural model checking on the discovered bounded partial state space. Furthermore, bounded model checkers usually do not support the full CTL [8]. Although there were theoretical results in this field, former bounded model checking approaches did not work well with CTL due to its branching semantics.

This paper extends our former work [11] described in Sect. 3.1 with new state space exploration strategies and with an efficient formula evaluation algorithm (namely constrained saturation) to traverse the bounded state space. This is the first approach where the constrained saturation is used in bounded model checking.

The structure of our paper is as follows: Sect. 2 introduces the background and prerequisites of our work. Sect. 3 gives an overview of the advanced saturation-based algorithms our work relies on. Sect. 4 describes the improvements of the bounded CTL model checking algorithm. Sect. 5 presents our measurement 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 *decision diagrams*, which form the underlying data structures of our algorithms.

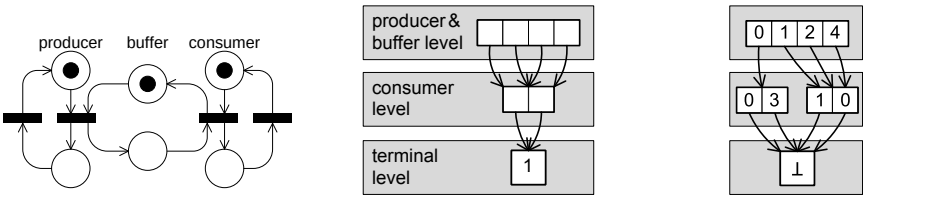
After that, we introduce the principles of bounded model checking. Finally, we overview the saturation-based state space exploration algorithm and the model checking background.

2.1 Decision Diagrams

This section is based on [11]. 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) [3] 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 nodes and terminal nodes (terminal 0 and terminal 1). The nodes are ordered into $K + 1$ levels. All *non-terminal nodes* are labelled by a variable index $1 \leq k \leq K$, which indicates the level the node belongs to (the variable it represents), and has n_k (domain size of the variable) arcs pointing to nodes in level $k - 1$. The *terminal nodes* are labelled by the variable index 0.

In our representation, duplicate nodes are not allowed, so if two nodes have identical successors in level k, they are also identical. These rules ensure that MDDs provide a canonical and compact representation of a given function or set. The function is evaluated by traversing the MDD top-down via the variable assignments represented by the arcs between nodes.

Figure 1(a) depicts a simple example Petri net [9] model of a producer-consumer system. The producer creates items and places them in the buffer, from where the consumer removes 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. Each edge encodes a possible local state [3]. 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 the producer-consumer model

(b) State space representation with MDD

(c) State space and distance representation with EDD

Figure 1: Producer-consumer example

The bounded saturation-based state space exploration algorithms use *Edge-valued Decision Diagrams* (EDDs) too. EDDs extend MDDs with extra informa-

tion: they can represent $f : \{0, 1, \ldots\}^K \to \mathbb{N} \cup \{\infty\}$ functions. It means that in addition to storing the state space, they can also store the distance information for bounded state space generation. As it is not directly used in our contributions, we omit the detailed introduction of EDDs and we refer the reader to [12] for details.

2.2 Model Checking and Bounded Model Checking

Given a formal model, *model checking* [7] is an automated technique to decide whether the model satisfies the specification. Formally: let M be a Kripke structure and let f be a formula of temporal logic. The goal of model checking is to find all states s of a model M such that $M, s \vDash f$ (meaning that the state s of model M satisfies f).

Bounded model checking [2, 6] decides whether the model satisfies the specification in a predefined number of steps, i.e., the depth of the state space traversal. Formally the problem for the k-bounded state space is to find all states s of M such that $M, s \vDash_k f$ (meaning that the state s of model M satisfies f in at most k steps). Among others, bounded model checking is useful when the full state space is not needed to decide on a property. This is e.g. the case for *shallow bugs*.

Structural model checking uses set operations to evaluate temporal logic specifications by computing fixed points in the state space. CTL [7] is a widely used temporal logic specifications formalism. It has expressive syntax, and structural model checking provides 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 is enough to express all the possible expressions due to duality [7].

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: $\mathbf{s} = (s_1, s_2, \ldots, s_n)$, 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 a limited set of components or just certain parts of the submodels. The algorithm divides the global next state function $\mathcal{N}$ into smaller parts: $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$, where $\mathcal{E}$ is the set of events in the high level model. The granularity of the decomposition can be chosen arbitrarily [5].

Saturation uses *symbolic encoding of the next state function*. In our work we use the symbolic next state representation from [1]. We represent $\mathcal{N}$ as a set of state

pairs, thus it can be stored in a decision diagram. For this set representation $\mathcal{R}$, the following is true: $\mathbf{s}' \in \mathcal{N}(\mathbf{s}) \Leftrightarrow (\mathbf{s}, \mathbf{s}') \in \mathcal{R} \Leftrightarrow \mathcal{R}(\mathbf{s}, \mathbf{s}')$ *is true*. The global relation can be expressed by the symbolic next state relations of the events: $\mathcal{R}(\mathbf{s}, \mathbf{s}') = \bigvee_{e \in \mathcal{E}} \mathcal{R}_e(\mathbf{s}, \mathbf{s}')$. Applying $\mathcal{N}_e$ to a given set of states represented by S results in $\mathcal{N}_e(S) = \mathit{RelProd}(\mathcal{R}_e, S)$, where *RelProd* is $\mathit{RelProd}(R, S) = \{s' | \exists s \in S, (s, s') \in R\}$ [5]. The smaller the partitions, the less computation they need, but there is a minimum imposed by the high-level 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 ($\mathcal{S}^{init}$). Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively, as if new states are discovered. Saturation iterates through the MDD nodes and generates the complete state space representation using a node-to-node transitive closure. 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.

Saturation-based Structural Model Checking. Saturation-based structural CTL model checking was first presented in [4], 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 [7] is the following:

- **EX**: $\mathbf{s}^0 \vDash \mathsf{EX}\ p$ iff $\exists \mathbf{s}^1 \in \mathcal{N}(\mathbf{s}^0)$ s.t. $\mathbf{s}^1 \vDash p$. EX corresponds to the function $\mathcal{N}^{-1}$, applying one step backward through the next state relation.
- **EG**: $\mathbf{s}^0 \vDash \mathsf{EG}\ p$ iff $\mathbf{s}^0 \vDash p$ and $\forall n > 0, \exists \mathbf{s}^n \in \mathcal{N}(\mathbf{s}^{n-1})$ s.t. $\mathbf{s}^n \vDash p$ so that there is a strongly connected component containing states satisfying p. This needs a greatest fixed point computation. Saturation cannot be applied directly, however the fixed point computation still benefits from the locality due to decomposition.
- **EU**: $\mathbf{s}^0 \vDash \mathsf{E}[p\ \mathsf{U}\ q]$ iff $\mathbf{s}^0 \vDash p$ and $\exists n > 0, \exists \mathbf{s}^1 \in \mathcal{N}(\mathbf{s}^0), \ldots, \exists \mathbf{s}^n \in \mathcal{N}(\mathbf{s}^{n-1})$ s.t. $\mathbf{s}^n \vDash q$ and $\mathbf{s}^m \vDash p$ for all $m < n$ (or $\mathbf{s}^0 \vDash q$). The states satisfying this property are computed with the following least fixed point: $\mathbf{lfp}\ Z[q \vee (p \wedge \mathsf{EX}\, Z)]$, i.e., we search for a state q reached through only states satisfying p.

3 Bounded and Constrained Saturation

In this section an overview is given of the two saturation-based advanced algorithms integrated in 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. Their integration leads to the 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 completely 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 this problem in the literature, both for globally and locally bounded saturation-based state space generation. We chose one that has already proved its efficiency [12]. Although MDDs provide a highly compact solution for state space representation, bounded saturation needs additional distance information during the traversal. For this reason, our chosen algorithm uses 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). The algorithm first iterates through the state space until a given bound is reached. 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. The algorithms in [12] use multiple different cutting strategies. In this work we use the strategy enforcing strict bounds (*TruncateExact* method) extended with our caching mechanism from [11].

3.2 Constrained Saturation

In [13] 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: in order to do the symbolic step $\mathcal{N}_e$ from the set of states S to a set of states satisfying the constraint C (represented by an MDD), we have to compute $\mathcal{N}_e(S) \cap C$. This contains an expensive intersection operation after each step. Using the following observation: $\mathcal{N}_e(S) \cap C = \mathit{RelProd}(\mathcal{R}_e, S) \cap C = \mathit{RelProd}(\{(\mathbf{s}, \mathbf{s}') | \mathcal{R}_e(\mathbf{s}, \mathbf{s}') \wedge \mathbf{s}' \in C\}, S)$, the algorithm can use pre-checking phase and avoid the computation-intensive intersection operation after the symbolic state space step [13], simply skipping the steps that "go out" of the constraint. Researches showed that the constrained saturation 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 Methods for Saturation-based Bounded Model Checking

In this section, a new, saturation-based bounded model checking algorithm is presented. The classical saturation-based, non-bounded model checking consists of

two consecutive steps: *1*) state space exploration; and *2*) evaluation of the CTL formula on the explored state space.

The bounded saturation-based model checking (see Fig. 2) performs these steps in an iterative way, where the state space exploration is done until a given bound b that is incremented in each iteration [11]. The input of this iterative algorithm is the initial bound B and an increment value δ: after each iteration, model checking analyses the property on the bounded state space and from the result it makes a *termination decision* if the iteration has to be continued. If the search is continued, then the procedure is restarted with an incremented depth $b := b + \delta$.

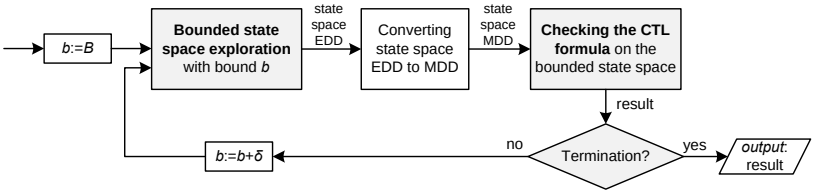


Figure 2: Overview of the saturation-based bounded model checking

Section 4.1 provides various methods for the *bounded state space exploration* using saturation. Section 4.2 describes a new, hybrid state space exploration approach combining EDDs and MDDs. Section 4.3 improves the CTL *formula checking* phase. Section 4.4 is dedicated to the discussion of the termination *decision mechanism*. Finally, Section 4.5 summarizes the contributions.

4.1 State Space Exploration Strategies

The used search strategy for the bounded model checking has a significant impact on performance. In this section different possible search strategy alternatives are introduced and compared. With regard to bounded state space generation, there are two main approaches: *(1)* Ideally, a fixed bound b can be given. The b-bounded state space is explored and the specification is evaluated on it. We call it the *fixed bound strategy*. This is typically not possible, as the necessary b is not known a priori. *(2)* Given an initial bound B and increment value δ, the state space is explored incrementally with bound $b = B + (n-1) \cdot \delta$ in each iteration n. The procedure stops when the model checking question is answered, or it runs out of resources. We call it the *incremental strategy*.

Traditionally, bounded model checking uses the incremental strategy, typically looking one step further ($\delta = 1$) in the state space in a breadth-first manner. According to the characteristics of the saturation iteration strategy, our experiences show 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 would lose the efficiency of saturation. On the other hand, a too large increase of search depth results in the loss of the efficiency of the bounded iteration strategy.

We have developed two basic incremental search strategies:

- The *restarting strategy* starts again from the initial state after each iteration, and uses the increased bound in the exploration [11].
- 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 is straightforward to implement and builds mainly on the efficiency of saturation. It simply uses the bounded saturation algorithm with growing bound values. We refer the reader to our previous work [11]. For the continuing strategy we had to modify the bottom-up building mechanism 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 contains 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 can reuse more from the formerly built data structures, making the algorithm typically more efficient than the restarting algorithm.

The result of the bounded state space exploration step is an EDD containing all the states that are within the current bound b, and the next-state relation $\mathcal{R}$ encoded by an MDD. These artefacts are exactly the same for both incremental strategies.

The next section describes a third, hybrid incremental search strategy, the compacting strategy.

4.2 Compacting Strategy

The two incremental state space exploration strategies introduced in the previous section have a common weakness: they store the whole $[0; b]$ part of the state space (i.e., the states with distance between 0 and b; marked as $\mathcal{S}_{[0;b]}$ in the following) in a single EDD. The algorithms use EDD for storing the distance information from the initial state for each state, which is not possible in case of the MDD encoding. This information is necessary to limit the state space exploration at the bound b. The storage of the distance has its price to pay: the EDD representation of the state space is typically less compact than the MDD representation (compare for example Fig. 1(b) and Fig. 1(c)).

While it is unavoidable to store distance information for some states to enforce the bound b, it is not necessary for all the states. Consider the nth iteration ($n > 1$). In this case, the $\mathcal{S}_{[0;b']}$ should be the result of the bounded state space exploration (where $b' = B + (n-1) \cdot \delta$). However, the $\mathcal{S}_{[0;b'-\delta]}$ part was already explored in the previous iteration, and it is known that these states will be present in $\mathcal{S}_{[0;b']}$ too. These states can be stored in the more compact MDD format. The main idea of the new so-called *compacting saturation* method is to store the state spaces discovered in previous iterations using a more compact, MDD-based representation.

Initial State Set. In the previous exploration strategies, each iteration was started either from the initial state (fixed bound strategy, restarting strategy), or from the state space of the previous iteration (that is $\mathcal{S}_{[0;b'-\delta]}$). The goal of compacting saturation is to avoid the EDD encoding for states $\mathcal{S}_{[0;b'-\delta]}$, if possible. Therefore this exploration cannot be started from the initial state of from $\mathcal{S}_{[0;b'-\delta]}$, as it would require the EDD encoding of these states.

It is easy to see that starting the algorithm from $\mathcal{S}_{[0;b'-\delta]}$ is unnecessary. Obviously, $\mathcal{N}(\mathcal{S}_{[0;b'-\delta-1]}) \subseteq \mathcal{S}_{[0;b'-\delta]}$. Therefore new states can only be found from the states on the "border of the bounded state space", i.e., from the states in $\mathcal{S}_{[b'-\delta;b'-\delta]}$, this will be the initial state set of the next iteration.

Details of the Compacting Saturation. In the following, the steps of the new compacting saturation algorithm are described in details.

The *first iteration* of the compacting saturation is started from the state (set) $\mathcal{I}_1 = \mathcal{S}^{init}$ and it explores the state space until the bound $b = B$. The result is the state set $\mathcal{S}_{[0;b]}$ encoded by EDD. Then the algorithm computes the border of the state space, i.e., $\mathcal{S}_{[b;b]} = \mathcal{C}_1$ and converts the EDD of $\mathcal{S}_{[0;b]}$ to an MDD representation $\mathcal{M} = \mathcal{M}_1$. The CTL formula will be evaluated on $\mathcal{M}$.

After, in *each iteration* $n > 1$ where the bound is $b' = B + (n-1) \cdot \delta$, the exploration is started from $\mathcal{I}_n = \mathcal{C}_{n-1} = \mathcal{S}_{[b'-\delta;b'-\delta]}$. The state space is explored until the bound b' that is $\mathcal{S}_{[b'-\delta;b']}$. Next, the algorithm computes the border of the state space $\mathcal{C}_n = \mathcal{S}_{[b';b']}$ and converts the EDD of the state space explored in the current iteration ($\mathcal{S}_{[b'-\delta;b']}$) to an MDD representation $\mathcal{M}_n$. The MDD $\mathcal{M}$ is modified in order to include $\mathcal{M}_n$ too, to represent $\mathcal{M} = \bigcup_{i=1}^{n} \mathcal{M}_i$. The evaluation of the CTL formula will be performed on this $\mathcal{M}$.

The state sets $\mathcal{C}_i$ can be computed using a modified version of the truncation operators described in [12].

Avoiding to Revisit States. During the traversal the previously explored states ($\mathcal{M}$) should not be re-explored: no states of $\bigcup_{i=1}^{n} \mathcal{M}_i$ should be explored again in iteration $n+1$. To be more precise, the goal of the algorithms is the following to keep: $\mathcal{M}_{n+1} \cap (\bigcup_{i=1}^{n} \mathcal{M}_i \setminus \mathcal{I}_{n+1}) = \emptyset$ (overlap in the initial state set is allowed). This is an obvious consequence of the state space parts defined above, i.e., $\mathcal{S}_{[b;b+\delta]} \cap \mathcal{S}_{[b+\delta;b+2\delta]} = \mathcal{S}_{[b+\delta;b+\delta]}$. However, if a state $\mathbf{s} \in \mathcal{M}_j$ $(j < n)$ is reachable from $\mathcal{I}_n$, it can be explored during the iteration n causing that it will be part of $\mathcal{M}_n$ too. It effectively means that the state $\mathbf{s}$ will be stored with two different distance value. The algorithm has to prevent this situation for efficiency reasons and also to keep the correctness.

A solution could be to subtract $\mathcal{M}$ from the state set of the current iteration after each step. It would make the method correct, but not efficient. The problem is similar to the motivation of the constrained saturation, except that here the forbidden states should be excluded, not the set of allowed states should be respected. Using the observation of constrained saturation, the following will be true: $\mathcal{N}_e(S) \setminus \mathcal{X} = \mathit{RelProd}(\mathcal{R}_e, S) \setminus \mathcal{X} = \mathit{RelProd}(\{(\mathbf{s}, \mathbf{s}') | \mathcal{R}_e(\mathbf{s}, \mathbf{s}') \wedge \mathbf{s}' \notin \mathcal{X}\}, S)$

If we use the observation above with $\mathcal{X} = \mathcal{M}$ as set of forbidden states, it can be efficiently avoided to reexplore states that were already explored in previous iterations.

4.3 Constrained Saturation using the Bounded State Space

Many model checking tools limit the syntax of the specification to a subset of the CTL temporal language, in order to simplify the analysis task and to boost the performance. We want to support the full semantics of CTL 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) [8].

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 approach would converge to traditional fixed point computations. Due to the symbolic representation of the saturation algorithm, it would be possible to check certain states that are not inside the explored, bounded part of the state space. It could be improved by constructing the intersection of the result of fixed point iterations with the bounded state space representation, but this still suffers from poor performance due to the extensive use of the costly intersection operation.

Our aim is to exploit the constrained saturation iteration strategy to provide an efficient bounded model checking algorithm. 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: instead of employing the intersection operation after each step (image computation), the algorithms apply intersection only at the beginning of the fixed point computations (when they compute the inputs of the constrained saturation algorithm). Below we define how the constrained saturation decides on the essential CTL operators (where **lfp** denotes the least fixed point, and *bss* denotes the bounded state space as stored by the MDD):

- **EF**: $M, s \vDash_k \mathsf{EF}\, p$ iff $s_0 \subseteq \mathbf{lfp}\ Z[(p \wedge bss) \vee (bss \wedge \mathsf{EX}\, Z)] = \mathit{ConsSaturation}(bss, p \cap bss)$[1]. This way we can directly exploit the constrained saturation algorithm to produce the least fixed point in the given bounded state space *bss*. The fixed point computational traversal explores only states inside *bss*, while the only intersection operation is used at the computation of the input argument: $p \cap bss$. 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 [13], but within a bounded setting. The fixed point computational traversal explores only states

[1] For the pseudocode of the *ConsSaturation* function we refer the reader to [13].

inside *bss*, while the only intersection operations are used at the computation of the input arguments: $bss \cap p$ and $bss \cap q$. 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) do 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.

4.4 Decision Mechanism

It is also necessary to be able to decide if an answer can be given to the requirement checked. The bounded model checking is a semi-decision procedure, therefore it can be used to ensure the following behavioural properties of the specification:

- *Invariant or safety*: proving these properties needs either the full state space to be explored, or bounded model checking can give a finite counterexample (or witness), if it exists.
- *Liveness*: bounded model checking can either find a *lasso-shaped* trace as counterexample (or witness) to these properties, or the full state space has to be explored to evaluate them.
- Other properties, like combination of safety and liveness properties: 3-valued logic can be used for decision. We refer the reader to our previous work [11].

Obviously, the bounded model checking can also be terminated if the full state space is already explored. In the case of the compacting algorithm, this can be detected easily by checking if $\mathcal{C}_n = \emptyset$. If $\mathcal{C}_n = \emptyset$, the algorithm terminates.

Invariant and safety properties are usually proved (by 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. The EDD-based state space representation helps us to tell more about liveness properties. Refuting liveness properties may come from the fact that: either (1) the algorithm has to explore more from the state space to find a witness, or (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 non-saturation-based 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 it only once to get this state set. It can be computed efficiently from the symbolic encoding. Let $result = \mathbf{lfp}\ Z[p_{d=bound} \vee (p \wedge \mathsf{EX}\,Z)] = ConsSaturate(p, p_{d=bound})$, thus *result* is

the set of states from which the $p_{d=bound}$ states can be reached only through states where p holds. If the initial state is not in the set *result*, the evaluation of $\mathsf{EG}\,p$ can be finished, as the result cannot be true: $s_0 \notin result \Rightarrow s_0 \not\models \mathsf{EG}\,p$.

4.5 Summary of the Contributions

In this section we described improvements of the saturation-based bounded model checking algorithm. New incremental state space exploration algorithms were presented (continuing and compacting saturation) that can reduce the overhead of the incremental methods compared to the fixed bound strategy.

With the novel use of constrained saturation, the algorithm avoids to examine states outside of the discovered bounded state space in the model checking phase without the need of expensive intersection operators. This way $\forall f(Z)$: $\mathbf{fp}\ f(Z) \subseteq bss$, for all fixed point the bounded saturation algorithm is bounded by the state space, even for the least fixed point computations.

5 Evaluation

We have performed many measurements in order to examine the efficiency of our new algorithm and compare it to a classical saturation-driven structural model checking algorithm. We have also examined the scalability of the new approach and compared to the former one. For this purpose we have developed an experimental implementation using the C# programming language of the fixed bound strategy (*B-Fix*), the restarting incremental strategy (*B-I-Rest*), the continuing incremental strategy (*B-I-Cont*), and for the compacting saturation (*B-I-Comp*). We have also implemented the algorithm taken from [13] as the reference for comparison, which we denoted in the measurements as "*Unlim*". For the measurements we used a desktop PC (Intel Core i7-3770 3.4 GHz CPU, 8 GB memory with Windows 7 x64 and .NET 4.0 framework).

5.1 Measurements on Benchmark Models

This section shows measurements on benchmark models widely known in the model checking community. We used the models[2] of Dining Philosophers (DPhil–N, [5]), Flexible Manufacturing System (FMS–N, [5]), Tower of Hanoi (Hanoi–N, [11]), an assembly line using Kanban method (Kanban–N, [5]), the Round Robin protocol (RR–N, [3]), and the Slotted Ring protocol (SR–N, [3]).

Both the initial bound B and the increment distance δ are input parameters, thus our algorithm can be fine-tuned by the user. If the properties to be proven 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

[2]The models can be downloaded from `http://petridotnet.inf.mit.bme.hu/publications/AC2014-Saturation_DarvasEtAl_models.zip`

proof in fewer iterations. A priori knowledge about the expected behaviour of the properties can significantly reduce the computational time.

Table 1 lists the runtime measurements for various CTL expressions on different models with different parameters.

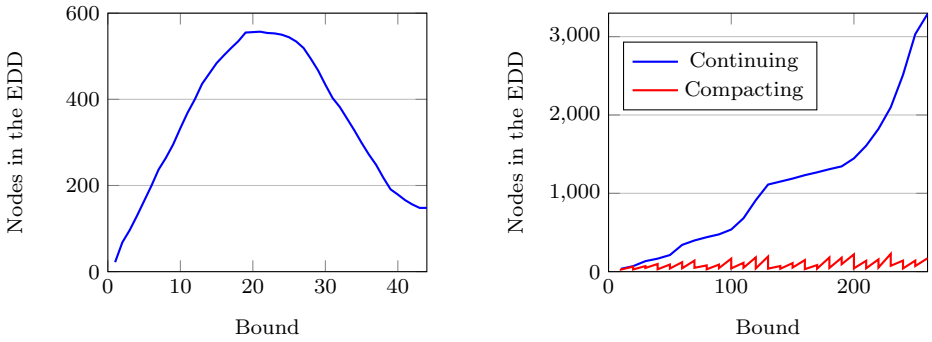
The Dining Philosophers (DPhil–N, where N is the number of philosophers) model revealed that for those models, where the saturation algorithm answers the model checking question quickly, the overhead of bounded model checking does not pay off. This is also the case for the chosen requirements for the Round Robin (RR-N) models, where the traditional saturation algorithm is highly efficient. These models are built from many identical components with relatively small local state spaces: the saturation algorithms traverses the possible state spaces very fast.

As can be seen in Table 1, saturation-based model checking can be highly efficient for asynchronous systems, and the modified bounded iteration strategy requires more computational resources, so one would expect that for these (especially for models DPhil and RR and SR) models the traditional approach performs better. However, the SR–N models (where N is the number of communication nodes) shows the advantage of bounded model checking, as traditional model checking has high runtimes even for a relatively simple property. This is a typical use case for bounded model checking as the property could be verified in a relatively small depth. The SR–N models have quite complex behaviour, where bounded model checking can significantly decrease the verification time.

We have measured the runtime for also synchronous models as the efficiency of the algorithms depends also on the extent of the synchronization in the systems. The verification goal was a combined safety-liveness property ($\mathsf{EG}(\mathsf{EF}(B_{\downarrow 8} = 1))$ of the Towers of Hanoi models, where $B_{\downarrow 8} = 1$ denotes the placement of the 8th disk to the 2nd rod, see the Petri net models for further details). The traditional structural model checking approach (Unbounded) runs out of resources early. Running the algorithm parameterised by the formerly computed bound (being enough to answer the verification question) (Bounded, fixed bound) has the best runtime results. The continuing strategy has advantage over the restarting strategy, as it uses up the formerly computed results during the model checking.

Similar results can be observed for the FMS model. The measurements of the FMS models also used a combined safety-liveness property that represents the existence of a cycle in a certain set of states satisfying safety requirements (based on [4]). The structural model checking algorithm time-outs for large parameters. For small values of N the unbounded method had the lowest runtime, but for increasing parameters the compacting saturation outperformed both the unlimited and the other bounded algorithms. The compacting method provided better results than the B-Fix method thanks to the efficient state space representation.

In some cases, like in the case of the FMS or the Kanban model with the chosen requirements, the restarting strategy solves the model checking problem for every parameter *faster* than the continuing strategy. We investigated the reason: as it is depicted in Figure 3(a), the state space representation of asynchronous systems (like FMS) grows steeply up to a maximal value (557 nodes), but after that it starts decreasing (resembling a bell curve) until 148 that is the final size of the state space



(a) FMS–3 (b) Hanoi–8

Figure 3: Size of state space representation (EDD) at each iteration

Table 1: Run times of CTL expression evaluation

	Run time[3] [s]				
N	**Unlim**	**B-I-Rest**	**B-I-Cont**	**B-I-Comp**	**B-Fix**
(1) DPhil–N, expression: $\mathsf{E}(\neg \mathit{eating}_2\ \mathsf{U}\ \mathit{eating}_1)$, $B = 20$, $\delta = 10$)					
10	**0.09**	0.10	0.09	0.33	*0.04*
50	**0.12**	5.02	5.12	1.81	*0.58*
100	**0.22**	12.78	12.92	3.88	*1.45*
(2) FMS–N, expression: $\mathsf{EG}(\mathsf{E}(\mathit{M1} > 0\ \mathsf{U}\ (\mathit{P1s} = \mathit{P2s} = \mathit{P3s} = 3)))$, $B = 20$, $\delta = 10$)					
25	**1.00**	40.02	40.57	18.65	*37.39*
50	**5.54**	56.31	59.51	21.96	*44.99*
100	48.33	61.68	63.51	**21.78**	*47.03*
1000	>600	60.17	62.12	**21.69**	*47.67*
10,000	>600	60.06	62.40	**21.59**	*54.34*
(3) Hanoi–N, expression: $\mathsf{EG}(\mathsf{EF}(B_{\downarrow 8} = 1))$, $B = 20$, $\delta = 10$)					
12	21.97	2.49	**0.81**	1.55	*0.36*
14	>600	2.89	**0.98**	2.39	*0.46*
16	>600	3.25	**1.11**	1.83	*0.49*
18	>600	3.60	**1.22**	1.91	*0.56*
20	>600	4.08	**1.43**	2.14	*0.67*
(4) Kanban–N, expression: $\mathsf{EF}(\mathit{pout}_4 = 5)$, $B = 20$, $\delta = 10$)					
30	**0.31**	1.01	1.29	3.11	*0.53*
50	1.91	**1.36**	1.73	3.65	*0.69*
100	26.51	**1.37**	1.78	3.64	*0.67*
200	>600	**1.37**	1.74	3.68	*0.68*
(5) RoundRobin–N (RR–N), expression: $\mathsf{EG}(\mathit{true})$, $B = 10$, $\delta = 5$)					
10	**0.04**	0.21	0.21	0.71	*0.05*
25	**0.31**	3.65	2.40	23.72	*1.29*
50	**2.05**	68.26	32.40	>300	*12.49*
(6) SlottedRing–N (SR–N), expression: $\mathsf{EF}(E_2 = 1 \wedge A_2 = 1)$, $B = 10$, $\delta = 5$)					
100	7.59	**0.89**	0.92	2.95	*0.84*
200	60.08	**2.31**	2.37	8.94	*2.29*
300	294.76	**4.31**	4.43	18.34	*4.30*

EDD. The continuing strategy uses these intermediate state space representations as the initial state, which can be 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 modifying an intermediate representation (i.e., using the continuing strategy) the algorithm has to do more computation, especially if the intermediate representation is larger than the final one.

The Kanban model shows the superiority of the bounded approaches compared to the full state space exploration. In addition, we must emphasize here that the new model checking algorithm is also superior to the former one of [11]. Constrained saturation-based model checking could reduce the runtime of the verification of properties from [11] on model Kanban–30 by approximately 70%. The reachability property in Table 1 required 90% less time to be verified by the new CTL model checking algorithm showing that we could efficiently utilize constrained saturation in bounded model checking.

We also analysed the EDD sizes of the Hanoi model (see Figure 3(b)) (representing synchronous models). In this case, the size of the state space representation grows constantly using the continuing strategy. However, the compacting saturation can effectively reduce the size of the EDD representation by storing the states discovered in previous iterations using MDD. That causes this sawtooth pattern.

5.2 Measurements on an Industrial Case Study

To evaluate our contribution we performed measurements on a model describing a real, industrial safety function. Our case study is a safety function included within the Reactor Protection System of a nuclear power plant [10][4]. This safety function initiates an emergency operation in case of a predefined chain of events happens. The detection of the specific event chain requires a complex logic, the design of which is error prone. This also puts emphasis on the necessity of using formal verification to ensure correctness.

The safety function receives inputs from different sensors, and computes the values of outputs, one of which initiates the emergency protection action. The values of the outputs depend on the recent and past values of the inputs, and some internal timers. The design of the controller was specified by simple combinatorial (OR gates, AND gates, and inverters), and sequential (SR flip-flops, delay and pulse modules) function blocks. The proper combination of these logic elements is required to guarantee that the emergency protection action will be initiated only in case of a specific dangerous event happened.

We have created a hierarchical coloured Petri net model of this safety logic. The structure of the CPN model preserves the data flow characteristics of the function block description. The subnets of the CPN model are the models for the functional modules. After the subnets have been derived and verified separately, they only had to be connected together properly [1]. The model can be parameterized, thus here

[3]The abbreviations used in the table headers are described in the first paragraph of Section 5.

[4]The authors of [10] decomposed the system and have done manual compositional verification.

Table 2: Run times of CTL expression evaluation on PRISE models

Model	Unlim	B-I-Rest	B-I-Cont	B-I-Comp
Expression 1: *OUTPUT-1 can be true.*				
PRISE S	2.84 s	**1.14** s	1.21 s	1.34 s
PRISE M	11.18 s	13.66 s	8.91 s	**4.79** s
PRISE L	27.88 s	13.72 s	8.98 s	**4.80** s
Expression 2: *There is no emergency action, if not necessary.*				
PRISE S	**5.37** s	33.97	19.98 s	12.18 s
PRISE M	**21.96** s	$>$ 1800 s	113.03 s	30.38 s
PRISE L	56.86 s	$>$ 1800 s	433.97 s	**51.83** s

we measured three different models with different complexity (denoted by PRISE S, M, and L).

The measurements being presented in Table 2 show that if the verification requirement is "shallow" (Expression 1), then the bounded algorithms provide significantly lower runtime than the unbounded algorithm, especially the compacting method. Expression 2 requires the exploration of a relatively big part of the state space. Therefore the bounded algorithms are slower than the unbounded algorithm. However, the compacting scales better and it provides slightly better runtime than the unbounded method for PRISE L. Also, the continuing strategy provides better results than the restarting strategy. Furthermore the restarting strategy runs out of memory for the PRISE M and L model, while the continuing and compacting strategy finish the execution for both models.

6 Conclusion and Future Work

In this paper an advanced bounded model checking approach based on the saturation algorithm was presented. It exploits the efficiency of saturation and enables us to verify complex, or in certain cases even infinite-state models. This approach also extends the set of asynchronous systems that can be analysed with symbolic methods. The efficiency of the new approach was proved by measurements.

We intend to develop the presented solutions further. We will investigate the use of forward model checking 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] Bartha, T., Vörös, A., Jámbor, A., and Darvas, D. Verification of an industrial safety function using coloured Petri nets and model checking. In *Proc. of the*

14th Int. Conf. on Modern Information Technology in the Innovation Processes of the Industrial Enterprises, pages 472–485. MTA SZTAKI, 2012.

[2] Biere, A., Cimatti, A., Clarke, E.M., and Zhu, Y. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[3] Ciardo, G., Marmorstein, R., and Siminiceanu, R. Saturation unbound. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 379–393. Springer, 2003.

[4] Ciardo, G. and Siminiceanu, R. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification*, volume 2725 of *LNCS*, pages 40–53. Springer, 2003.

[5] Ciardo, G., Zhao, Y., and Jin, X. Ten years of saturation: A Petri net perspective. In *Transactions on Petri Nets and Other Models of Concurrency V*, volume 6900 of *LNCS*, pages 51–95. Springer, 2012.

[6] Clarke, E.M., Biere, A., Raimi, R., and Zhu, Y. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

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

[8] Henzinger, T., Kupferman, O., and Qadeer, S. From pre-historic to post-modern symbolic model checking. In *Computer Aided Verification*, volume 1427 of *LNCS*, pages 195–206. Springer, 1998.

[9] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[10] Németh, E. and Bartha, T. Formal verification of safety functions by reinterpretation of Functional Block based specifications. In *Formal Methods for Industrial Critical Systems*, volume 5596 of *LNCS*. Springer, 2009.

[11] Vörös, A., Darvas, D., and Bartha, T. Bounded saturation-based CTL model checking. *Proceedings of the Estonian Academy of Sciences*, 62(1):59–70, 2013.

[12] Yu, A., Ciardo, G., and Lüttgen, G. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *International Journal on Software Tools for Technology Transfer*, 11:117–131, 2009.

[13] Zhao, Y. and Ciardo, G. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Automated Technology for Verification and Analysis*, volume 5799 of *LNCS*, pages 368–381. Springer, 2009.

Received 10th March 2014