Saturation-based Incremental LTL Model Checking with Inductive Proofs

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

[1] Budapest University of Technology and Economics, Hungary
[2] Institute for Computer Science and Control, Hungarian Academy of Sciences

Abstract. Efficient symbolic and explicit model checking approaches have been developed for the verification of linear time temporal properties. Nowadays, advances resulted in the combination of on-the-fly search with symbolic encoding in a hybrid solution providing many results by now. In this work, we propose a new hybrid approach that leverages the so-called saturation algorithm both as an iteration strategy during the state space generation and in a new incremental fixed-point computation algorithm to compute strongly connected components (SCCs). In addition, our solution works on-the-fly during state space traversal and exploits the decomposition of the model as an abstraction to inductively prove the absence of SCCs with cheap explicit runs on the components. When a proof cannot be shown, the incremental symbolic fixed-point algorithm will find the SCC, if one exists. Evaluation on the models of the Model Checking Contest shows that our approach outperforms similar algorithms for concurrent systems.

1 Introduction

Linear temporal logic (LTL) specifications play an important role in the history of verification. Checking these properties is usually reduced to finding *strongly connected components* (SCCs) by checking language emptiness of the synchronous product of two Büchi automata: one characterizing the possible behaviors of the system and another accepting behaviors that violate the desired property. Two main approaches emerged during the history of model checking. *Explicit* methods process the state graph using proven graph algorithms. *Symbolic* model checking was introduced to address the problem of state space explosion. Symbolic approaches based on *decision diagrams* usually apply greatest fixed point computations on the set of states to compute an *SCC-hull* [14]. These approaches typically scale well, and they have improved considerably due to the extensive research in this area.

A considerable amount of effort was put in combining symbolic and explicit techniques [1, 10–13]. The motivation is usually to introduce one of the main advantages of explicit approaches into symbolic model checking: the ability to look for SCCs on the fly, i. e., continuously during state space generation. Solutions typically include abstracting the state space into sets of states such as in the

Author's manuscript.
The original publication is available at www.springerlink.com.

Published in: C. Baier and C. Tinelli, editors, Tools and Algorithms for the Construction and Analysis of Systems, volume 9035 of Lecture Notes in Computer Science, pp. 643-657. Springer-Verlag, 2015.
DOI: 10.1007/978-3-662-46681-0_58

case of multiple state tableaux or symbolic observation graphs. Explicit checks can then be run on the abstraction on the fly to look for potential SCCs.

The goal of this paper is to present a new hybrid LTL model checking algorithm that *1)* builds a symbolic state space representation, *2)* looks for SCCs on the fly, *3)* incrementally processes the discovered parts of the state space and *4)* uses explicit runs on multiple fine-grained abstractions to avoid unnecessary computations. Although example models are given as Petri nets, the algorithm can handle any discrete state model. The state space is encoded by decision diagrams, built using saturation. On-the-fly detection of SCCs is achieved by running searches over the discovered state space continuously during state space generation. In order to reduce the overhead of these searches, we present a new incremental fixed point algorithm that considers newly discovered parts of the state space when computing the SCC-hull. While this approach specializes on *finding* an SCC, a complementary algorithm maintains various abstractions of the state space to perform explicit searches in order to inductively prove the *absence* of SCCs.

The paper is structured as follows. Section 2 presents the background of this work. An overview of the proposed algorithm is given in Section 3, then Section 4 and 5 introduces the main components in detail. The whole algorithm is assembled in Section 6. A brief summary of related work is presented in Section 7, followed by an extensive evaluation of our approach and three other tools in Section 8. Finally, Section 9 summarizes the contributions of the paper.

2 Saturation

Saturation is an iteration strategy specifically designed to work with decision diagrams. It was originally used as a state space generation algorithm [5] to answer reachability queries on concurrent systems, but applications in branching-time model checking [17] and SCC computation [18] also proved to be successful.

Saturation works best if it can exploit the structure of high-level models. Therefore, it defines the input model on a finer level of granularity, introducing the concept of components and events into traditional discrete-state models. Formally, the input model of the algorithm is in the form $M = \langle \mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N} \rangle$. Provided that the model has K components, each with the set of possible *local states* $\mathcal{S}_k$, we call $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_K$ the set of possible *global states*. A single global state $\mathbf{s}$ is then a K-tuple $(s_1, \ldots, s_K)$, where each $s_k \in \mathcal{S}_k$ is a state variable containing the local state of the kth component. The set of possible initial states of the system is $\mathcal{S}_{init} \subseteq \mathcal{S}$. Elements of set $\mathcal{E}$ are (asynchronous) *events* of the model, usually corresponding to transitions of the high-level system model. Events are used to decompose the next-state (or *transition*) relation $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ into separate (but not necessarily disjoint) next-state relations: $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$, where $\mathcal{N}_\varepsilon$ is the next state relation of event ε. We often use $\mathcal{N}$ as a function, defining $\mathcal{N}(\mathbf{s}) = \{\mathbf{s}' \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}\}$ as the set of states that are reachable from $\mathbf{s}$ in one step (and also $\mathcal{N}(S)$ as an extension to sets of states). The inverse of a next state function is defined as $\mathcal{N}^{-1}(\mathbf{s}) = \{\mathbf{s}' \mid (\mathbf{s}', \mathbf{s}) \in \mathcal{N}\}$. In this paper, a

state space will be denoted by a pair $(\mathcal{S},\mathcal{N})$, where states of $\mathcal{S}$ and transitions of $\mathcal{N}$ are nodes and arcs of the state graph.

By introducing components and events, saturation can exploit the *locality* property of concurrent systems. Locality is the empirical assumption that high-level transitions of a concurrent model usually affect only a small number of components. An event ε is *independent* from component k if *1)* its firing does not change the state of the component, and *2)* its enabling does not depend on the state of the component. If ε depends on component k, then k is called a *supporting* component: $k \in \mathit{supp}(\varepsilon)$.

In order to map the components to variables of the underlying decision diagram, an ordering has to be defined. Without loss of generality, assume that every component is identified by its index in the ordering. Using these indices, it is possible to group events by defining $\mathit{Top}(\varepsilon) = k$ as the supporting component of ε with the highest index. The set of every event with a *Top* value of k is $\mathcal{E}_k = \{\varepsilon \in \mathcal{E} \mid \mathit{Top}(\varepsilon) = k\}$. For the sake of convenience, we use $\mathcal{N}_k$ to represent the next state function of all such events, formally $\mathcal{N}_k = \bigcup_{\varepsilon\in\mathcal{E}_k} \mathcal{N}_\varepsilon$. The notations $\mathcal{N}_{\leq k} = \bigcup_{i\leq k} \mathcal{N}_i$ and $\mathcal{N}_{<k} = \bigcup_{i<k} \mathcal{N}_i$ will also be used.

Symbolic encoding of the next state functions of events $\varepsilon \in \mathcal{E}_k$ relies on the following observation: $\mathcal{N}_\varepsilon((s_1,\ldots,s_K))$ and $\mathcal{N}_\varepsilon((s_1,\ldots,s_k)) \times \{(s_{k+1},\ldots,s_K)\}$ are equivalent (i. e., $\mathcal{N}_\varepsilon$ does not change the local states of components above k). From this fact, two important properties of saturation follows: *1)* in the encoding of $\mathcal{N}_\varepsilon$ it is sufficient to encode the state changes of state variables $s_1,\ldots,s_k$, where $k = \mathit{Top}(\varepsilon)$, as well as *2)* it is possible to apply the individual $\mathcal{N}_\varepsilon$ functions in a finer granularity: $\mathcal{N}_\varepsilon$ is not only applicable on a set of global states, but also on sets of substates composed of state variables $s_1,\ldots,s_k$.

In order to reason about sets of substates encoded by decision diagram nodes, we will use the notations introduced in [4]. Let n_k be a single node in a decision diagram on the level representing the state variable of the kth component. Let $\mathcal{B}(n_k)$ represent the *below* substates encoded by n_k. Below substates can be regarded as the set of paths in the decision diagram that go from n_k to the terminal node $\mathbf{1}$. Throughout this paper, $n_k[i]$ will denote the *child node* of n_k on level $k-1$ reachable through the arc corresponding to the local state $i \in \mathcal{S}_k$. With this notation, the set of substates $\mathcal{B}(n_k)$ encoded by node n_k is described with the following recursive definition:

$$\mathcal{B}(n_k) = \begin{cases} \{i \mid n_k[i] = \mathbf{1}\} & \text{if } k = 1 \\ \bigcup_{i\in\mathcal{S}_k} \mathcal{B}(n_k[i]) \times \{i\} & \text{otherwise.} \end{cases}$$

A possible interpretation of the definition is that the set of substates encoded by node n_k is composed of different instantiations of the sets of substates encoded by the children of n_k.

The goal of saturation as a state space generation algorithm is to compute the set of reachable states $\mathcal{S}_{rch} = \mathcal{N}^*(\mathcal{S}_{init})$ of model M, where $\mathcal{N}^*$ is the *transitive closure* of the next-state relation. To do this, it exploits the structure of decision diagrams and the aforementioned locality of concurrent systems by dividing the global fixed-point computation into smaller parts, computing local

fixed-points with regard to a decision diagram node n_k and its corresponding next-state function $\mathcal{N}_k$. A node n_k is called *saturated*, if it is a terminal node, or its child nodes are saturated and it represents a set of substates computed as the fixed-point of the transitive closure of $\mathcal{N}_k$, formally: $\mathcal{B}(n_k) = \mathcal{N}^*_{\leq k}(\mathcal{B}(n_k))$. This definition yields a recursive algorithm that saturates nodes of the decision diagram in a bottom-up order, recursively saturating new nodes discovered when applying a next-state function on higher levels of the decision diagram.

3 Overview of the Algorithm

The goal of this paper is to present a new model checking solution that is *1)* symbolic, *2)* looks for SCCs on the fly during state space generation with an incremental fixed-point algorithm, and *3)* uses cheap explicit proofs to indicate the absence of SCCs when possible. The basis of the presented complex algorithm is saturation, which is highly efficient in the symbolic state space generation of large concurrent systems.

On-the-fly operation is achieved by performing fixed-point computations *when a node becomes saturated*. Processing saturated nodes has the advantage of handling a set of (sub)states that is closed with regard to events independent from higher levels. This means that the set will not change anymore during the exploration, i. e., each closed set has to be processed only once.

Even though a set with its related events will be processed only once, the recursive definition of saturation will cause such sets to appear again as part of larger sets encoded by the parent node in the decision diagram. The incremental fixed-point algorithm presented in Section 4 avoids redundant computations by restricting the search to SCCs containing at least one transition belonging to an event not considered before, causing the computation to converge faster.

It has been shown many times that symbolic model checking approaches can greatly benefit from explicit techniques [1, 10–13]. In this work, explicit checks are applied in two ways. First, the saturation algorithm is enhanced with a simple modification that is able to collect individual states appearing more than once during the exploration. As presented in Section 5.1, the absence of these *recurring* states indicates that no SCCs can be found in the set of explored states. Secondly, one of the main contributions of this paper is a cheap abstraction of the state space with regard to a single decision diagram node, on which explicit SCC computation algorithms can be run with a negligible overhead. A theorem presented in Section 5.2 gives an efficient method to inductively prove the absence of SCCs in the state space explored so far. Both methods are used to reduce the number of times a symbolic fixed-point computation is necessary, often making the overhead of on-the-fly searches to almost disappear.

4 Incremental Symbolic Fixed-point Computation

This section presents a symbolic fixed-point computation algorithm to look for SCCs incrementally in a growing state space. It can be regarded as a variation

of traditional SCC-hull algorithms [14], but it is unique in the sense that it is optimized to run multiple times, each time on a superset of the previous input. SCC-hull algorithms usually start with a set of states and a transition relation and iteratively try to discard states to reach a fixed-point. Compared to this strategy, the main difference in our concept is that we specify *transitions* to discard. The reason for this design lies in the iteration strategy of saturation, but the algorithm itself is not restricted to any iteration strategy.

As noted in Section 2, the set of substates encoded by a node n_k can be written as $\mathcal{B}(n_k) = \bigcup_{i \in \mathcal{S}_k} \mathcal{B}(n_k[i]) \times \{i\}$, i. e., the union of the below substates of each child node instantiated with the corresponding value of the kth state variable. In case of a saturated node, each set in the union is closed with regard to the next-state function $\mathcal{N}_{<k}$, so no new SCCs can be found in $\mathcal{B}(n_k)$ using these transitions only. However, the sets of substates are connected by transitions in $\mathcal{N}_k$ that are not yet processed on $\mathcal{B}(n_k)$. Figure 1 shows an example: black arcs between sets of substates are transitions of $\mathcal{N}_k$. Constraining the search for SCCs to those that contain at least one transition from $\mathcal{N}_k$ can quickly discard parts of the state space to make the fixed point computation converge faster.

The main function of the algorithm, *DetectSCC*, does not have to know about saturation. It takes a set of states ($\mathcal{S}$), a next-state relation ($\mathcal{N}$), and a subset of this relation ($\mathcal{N}_{new} \subseteq \mathcal{N}$) as an input and returns a Boolean value indicating if there exists an SCC consisting of states and transitions from $\mathcal{S}$ and $\mathcal{N}$ that contains at least one transition from $\mathcal{N}_{new}$. The following observation provides a way to use this function as an incremental SCC computation algorithm.

Observation 1 *Consider a state space $(\mathcal{S}, \mathcal{N})$ containing an SCC and a subset of transitions $\mathcal{N}_{new} \subseteq \mathcal{N}$ considered to contain new transitions. If $\mathcal{S}$ does not contain any SCCs with only old transitions $\mathcal{N}_{old} = \mathcal{N} \setminus \mathcal{N}_{new}$, then the SCC contains at least one transition from $\mathcal{N}_{new}$.*

Let us assume that the function is called during state space exploration and the input is the current set of (sub)states, the set of transitions fired so far, and the set of transitions fired since the last time the algorithm was called. Then, the observation guarantees that by the end of the state space generation, the algorithm will have returned *true* at least once iff there exists an SCC in the reachable state space $(\mathcal{S}_{rch}, \mathcal{N}_{fired})$. The case of calling the function in a recursive setting is less obvious. To see that calling *DetectSCC* after a node is saturated gives a complete algorithm, consider the previous discussion. When a node n_k on level k becomes saturated, the only transitions that were fired but have never been input into the function yet are in $\mathcal{N}_k$. This way, the correct inputs when calling the function at that point are $\mathcal{B}(n_k)$ as the set of states, $\mathcal{N}_{\leq k}$ as the next-state relation and $\mathcal{N}_k$ as the subset of new transitions.

Algorithm 1 shows the pseudocode of *DetectSCC*. The function works by discarding transitions from $\mathcal{N}_{new}$ that cannot be closed with other transitions through the states in $\mathcal{S}$ to form a loop. Transitions are not processed directly: the set of their *source* states $\mathcal{S}^-$ and *target* states $\mathcal{S}^+$ represent them in the fixed-point computation. States of $\mathcal{S}^-$ that are not reachable from $\mathcal{S}^+$ and states of

input : $\mathcal{S}, \mathcal{N}, \mathcal{N}_{new}$: set
// $\mathcal{S}$: set of states,
// $\mathcal{N}, \mathcal{N}_{new}$: set of transitions
output : bool

1 $\mathcal{S}^- \leftarrow \mathcal{N}_{new}^{-1}(\mathcal{S})$; $\mathcal{S}^+ \leftarrow \mathcal{N}_{new}(\mathcal{S}^-)$;
2 **if** $\mathcal{S}^+ = \emptyset$ **then return** *false*;
3 **repeat**
4 $\quad \mathcal{S}^- \leftarrow \mathcal{S}^- \cap \mathcal{N}^*(\mathcal{S}^+)$;
5 $\quad \mathcal{S}^+ \leftarrow \mathcal{S}^+ \cap \mathcal{N}_{new}(\mathcal{S}^-)$;
6 **until** $\mathcal{S}^+$ *and* $\mathcal{S}^-$ *unchanged*;
7 **return** $\mathcal{S}^- \neq \emptyset \vee \mathcal{S}^+ \neq \emptyset$;

Algorithm 1. DetectSCC

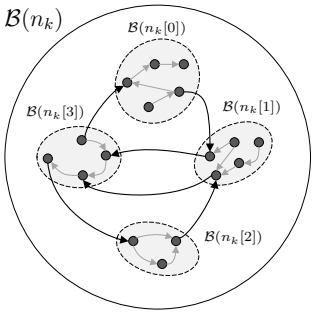


Fig. 1. Illustration of state space $(\mathcal{B}(n_k), \mathcal{N}_{\leq k})$.

$\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ *through transitions in* $\mathcal{N}_{new}$ are discarded iteratively in lines 4 and 5. Checking reachability is performed using saturation. The iteration stops when no states can be discarded from the sets anymore, i. e., fixed point is reached. If $\mathcal{S}^-$ and $\mathcal{S}^+$ are empty, then no appropriate SCC could be found. Otherwise, the remaining states are part of an SCC containing at least one transition from $\mathcal{N}_{new}$. Since the goal of this algorithm is to quickly decide if an SCC exists in the input, it *will not* extract the SCC itself. However, remaining states in $\mathcal{S}^+$ and $\mathcal{S}^-$ can be used to aid the counterexample generation.

When looking for *fair SCCs*, i. e., SCCs containing at least one state from a set of states $\mathcal{F}$, the algorithm can be extended to involve $\mathcal{F}$ as the third set in the loop. States of $\mathcal{F}$ are then discarded if they are not reachable from $\mathcal{S}^+$ and states of $\mathcal{S}^-$ are discarded if they are not reachable from $\mathcal{F}$. When looking for accepting SCCs during LTL model checking, $\mathcal{F}$ is the set of accepting states.

5 Explicit Proofs

After presenting an incremental way to detect the presence of strongly connected components during state space generation, this section introduces methods to prove the absence of SCCs without performing symbolic fixed point computations. These methods are used to decide if a symbolic check should be performed when a node is saturated or it can be safely omitted.

When looking for accepting SCCs, checking the absence of accepting states is a usual optimization in similar algorithms, for example in the abstraction refinement approach presented in [16]. In this paper, we go two steps further. Section 5.1 introduces the use of recurring states, while Section 5.2 presents a new abstraction technique tailored to decision diagrams that allows the direct use of explicit algorithms to reason about the presence or absence of SCCs.

5.1 Using Recurring States for Explicit Proofs

Recurring states are those that have already been discovered before reaching them again during state space generation. In the context presented in Section

4, they are defined as follows: $\mathcal{R} = \mathcal{S}_{old} \cap \mathcal{N}_{new}(\mathcal{S}_{old})$, where $\mathcal{S}_{old}$ is the set of discovered states before applying $\mathcal{N}_{new}$ for the first time. Explicit SCC computation algorithms such as [15] primarily look for recurring states during graph traversal as they are suspects to constitute SCCs. Checking backward reachability from these states offers a simple algorithm to check the presence of an SCC [8]. Symbolic algorithms, on the other hand, execute many steps together, making the individual checking of the states inefficient. However, computing the set of recurring states during state space traversal can still be used to reason about SCCs.

Observation 2 *Given an SCC composed of a set of states $\mathcal{S}$ and a next-state relation $\mathcal{N}$, any traversal will yield at least one recurring state.*

According to the observation, recurring states offer a cheap way to distinguish situations where there is no chance of finding an SCC – situations that often arise during an on-the-fly algorithm. In addition, they can also be used to initialize the fixed-point computation algorithm with $\mathcal{S} := \mathcal{R}$ in Algorithm 1. This is useful if recurring states are collected between two subsequent *DetectSCC* calls, because *1)* only transitions of $\mathcal{N}_{new}$ can end in recurring states and *2)* this way, the function can also exploit Observation 2 and restrict the search for SCC candidates containing new recurring states.

5.2 Introducing Inductive Explicit Checks

Hybrid model checking algorithms usually use symbolic encoding to process huge state spaces, accompanied by clever abstraction techniques to produce an abstract model on which explicit graph algorithms can be used. In this context, the goal of abstraction is to reduce the size of a system's state space while preserving certain properties, such as the presence or absence of SCCs. In this work, we also use abstractions to reason about SCCs. However, unlike in most approaches in this domain, multiple abstract state graphs are used, ordered in a hierarchy matching the structure of the underlying decision diagram to build an inductive proof about strongly connected components of the state space.

In a symbolic setting, components of the model provide a convenient basis for abstraction. In LTL model checking, it is usual to use the Büchi automaton or its observable language to group states and build an abstraction from these aggregates. The abstraction framework presented in [16] goes beyond using only one kind of abstraction and explores strategies on a tableau of possible abstractions based on one or more components.

In addition to selecting the basis, there are multiple ways to define an abstraction based on a component. To illustrate this, two simple abstractions are presented before introducing a new approach of using the structure of a decision diagram to define a more powerful abstraction.

Simple Abstractions. Using abstractions to answer binary decisions has two potential goals. One can create an abstraction that can say a definite *yes* (these

are called *must abstractions*), or one that can say a definite *no* (these are *may abstractions*). To construct an abstraction, the definition of an abstraction function is required for both the states and the transitions in the global state space. Abstracting states is straightforward, as the set of local states $\mathcal{S}_k$ of component k can be used directly.[3] Regarding may and must abstractions, different transformations have to be defined for the transitions of the state space.

Must abstraction of transitions $\mathcal{N}_k^{\forall} \subseteq \mathcal{S}_k \times \mathcal{S}_k$ for component k is defined as: $\mathcal{N}_k^{\forall} = \{(s_k, s'_k) | \exists \varepsilon \in \mathcal{E}, supp(\varepsilon) = \{k\}, \exists (\mathbf{s}, \mathbf{s}') = ((\dots, s_k, \dots), (\dots, s'_k, \dots)) \in \mathcal{N}_\varepsilon\}$. *May abstraction* of transitions $N_k^{\exists} \subseteq S_k \times S_k$ for component k is defined as $\mathcal{N}_k^{\exists} = \{(s_k, s'_k) | \exists \varepsilon \in \mathcal{E},\ k \in supp(\varepsilon), \exists (\mathbf{s}, \mathbf{s}') = ((\dots, s_k, \dots), (\dots, s'_k, \dots)) \in \mathcal{N}_\varepsilon\}$. The must abstraction of transitions is defined to keep only those transitions that correspond to events fully within the support of the chosen component. May abstraction preserves every local transition, but omits the synchronization constraints (i. e., assumes that if a transition is enabled in component k, it is globally enabled).

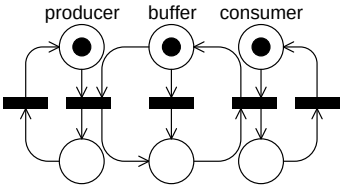
Due to this construction, it is sometimes possible to reason about the presence or absence of global SCCs. If there is an SCC in a single must abstraction, it is the direct representation of one or more SCCs of the global state space. Complementary, if there is no SCC in the may abstraction of *any* component, then the global state space cannot contain any SCCs either.

These abstractions usually yield small state graphs that can be represented explicitly. Running linear-time explicit algorithms on them gives a very cheap opportunity to possibly prove or refute the presence of SCCs before symbolic methods are used. Moreover, the definition of may and must abstractions implies $\mathcal{N}_k^{\forall} \subseteq \mathcal{N}_k^{\exists}$, so running the SCC computation on a may abstraction and then looking for a strongly connected subcomponent with transitions of $\mathcal{N}_k^{\forall}$ effectively considers both cases at the same time.

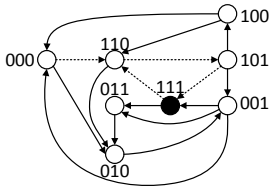
As an example, observe Figure 2 that illustrates the Petri net model of a producer-consumer system, also showing the explicit state graph. Transitions of the system are shown on Figure 3(a), with connected arcs representing a single transition affecting multiple components. In this case, every transition belongs to a separate event (events are related to transitions of the Petri net). Events affecting multiple components can be regarded as synchronization constraints between *local transitions*. Abstractions can be acquired by removing synchronizations and local transitions. Figure 3(b) and 3(c) depict the transitions transformed by must and may abstractions. If the goal is to find an SCC containing the state where only the places at the bottom of the Petri net are marked (depicted as a black state on Figure 2(b)), none of the abstractions can give an exact answer.

Node-wise Abstraction. As the example suggests, the simple abstractions presented so far may often be too general/specific, limiting their usefulness. Also, as before, the iteration strategy of saturation can be exploited when designing a special type of may abstraction that is stronger than its simple version. The goal

[3] It is assumed that local states in $\mathcal{S}_k$ actually appear in at least one reachable global state.

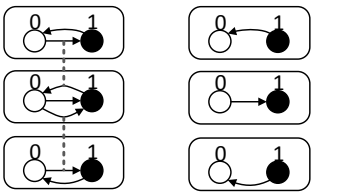


(a) Petri net

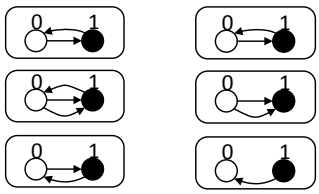


(b) State space

Fig. 2. Producer-consumer model with non-deterministic buffer



(a) Original (b) Must



(c) May (d) Node-wise

Fig. 3. The effect of the abstractions to the transitions

of the following construct is to match the order in which events are processed during saturation, as well as the structure of the underlying decision diagram.

Definition (Node-wise abstraction) *Node-wise abstraction of state space* $(\mathcal{S},\mathcal{N})$ *with regard to node* n_k *is the pair* $\mathcal{A}^{\exists}_{n_k} = (\mathcal{S}_{n_k},\mathcal{N}^{\exists}_{n_k})$*, where* $\mathcal{S}_{n_k} = \{i \mid n_k[i] \neq \mathbf{0}\}$*, i. e., the local states encoded by arcs of* n_k*, and* $\mathcal{N}^{\exists}_{n_k} = \{(s_k, s'_k) \mid s_k, s'_k \in \mathcal{S}_{n_k}, \exists(\mathbf{s},\mathbf{s}') = ((\ldots,s_k,\ldots),(\ldots,s'_k,\ldots)) \in \mathcal{N}_k\}$*, i. e., the projections of events* $\mathcal{E}_k$ *to component* k*.*

By the time a node is saturated, the construction of its node-wise abstraction is permanently finished. This way, a single abstraction has to be analyzed only once. In addition, the set of node-wise abstractions corresponding to nodes of a sub-diagram rooted in n_k contains enough information to have the power of the simple may abstraction, that is, to clearly state if no SCC is present in the substate space $(\mathcal{B}(n_k),\mathcal{N}_{\leq k})$.

The following theorem gives the basis for an inductive method of using node-wise abstractions to prove the absence of SCCs.

Theorem *Given a node-wise abstraction* $\mathcal{A}^{\exists}_{n_k}$ *with regard to a saturated node* n_k*, the substate space* $(\mathcal{B}(n_k),\mathcal{N}_{\leq k})$ *does not contain any SCC if 1) neither of the substate spaces* $(\mathcal{B}(n_k[s_i]),\mathcal{N}_{<k})$ *belonging to the children of* n_k *2) nor* $\mathcal{A}^{\exists}_{n_k}$ *contain an SCC.*

The main idea of the proof is that node-wise abstraction represents the effects of the events $\mathcal{E}_k$ exactly on the level of their *Top* value. At the time when a node n_k becomes saturated, the only transitions that can change the local states of component k are in $\mathcal{N}_k$. Node-wise abstractions contain the images of exactly

these transitions, thus they describe the possible transitions between sets of substates encoded by the children of n_k, as seen on Figure 1. This is why they can be used to identify *one-way walls* that separate the possible spaces for SCCs. Figure 1 can be seen as a node-wise abstraction if gray sets are considered as states of $\mathcal{A}^{\exists}_{n_k}$, with black arcs between them being transitions of $\mathcal{N}^{\exists}_{n_k}$.

Note that it is not specified how to ensure assumption 1 of the theorem. Consequently even if the corresponding node-wise abstraction did contain an SCC (which only implies the *possible* presence of a global SCC), the symbolic fixed point computation algorithm of Section 4 can still be used to give a precise proof. This way, the series of saturated nodes give a full inductive proof by the end of the state space generation. In the previous example shown on Figure 3, node-wise abstraction *can* predict that SCC detection is unnecessary during saturation until the top level is processed.

The computation of node-wise abstractions is simple and cheap. It can be done on demand by projecting the next-state relation of corresponding events to the *Top* component, or on-the-fly during saturation by adding vertices and arcs each time a new local state is discovered or a new transition of the corresponding events is fired, respectively. A simple must abstraction can also be examined as part of computing SCCs of the node-wise abstraction by looking for a strongly connected subcomponent whose transitions belong to events having only the current component as a supporting one.

In addition to proving the absence of SCCs, the result of explicit computation on the abstraction can also be used to aid the incremental fixed point computation algorithm in finding them. Arcs of the *candidate SCCs* found in the node-wise abstraction correspond to a set of transitions in the state space ($\mathcal{N}_{SCC}$). Since these are the *only* transitions in $\mathcal{N}_k$ that can be part of an SCC, calling *DetectSCC* with $\mathcal{N}_{new} = \mathcal{N}_{SCC}$ helps the function to converge even faster.

6 Constructing the Algorithm

After getting familiar with the building blocks in Sections 2–5, this section assembles the main contribution of this work, the new saturation-based incremental LTL model checking algorithm. The algorithm uses saturation for state space generation. Recurring states are collected on the fly and vertices and arcs of node-wise abstractions may also be added continuously. Whenever a node becomes saturated, the following steps are executed:

1. The sets of encoded states and transitions are checked (shall be non-empty).
2. The set of collected recurring states is checked (shall be non-empty).
3. An explicit SCC computation algorithm is run on the current node-wise abstraction to obtain an SCC candidate (there shall be one).
4. Function *DetectSCC* is called with the set of recurring states and transitions in the candidate SCC to compute an SCC-hull.

If either of checks 1–3 fails, or *DetectSCC* returns *false*, saturation continues. If at any point *DetectSCC* returns *true*, the algorithm is stopped and the LTL

```
   input    : s_k : node      // to saturate
   output   : node
 1 n_2k ← N_k as decision diagram;
 2 t_k ← new node;   A^∃_{t_k} ← (S_k, ∅);
 3 foreach i ∈ S_k : s_k[i] ≠ 0 do
 4 |  t_k[i] ← Saturate(s_k[i]);
*5 r_k ← new node;                 // recurring states
 6 repeat
 7 |  foreach i, i' ∈ S_k : s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*8 |  |  r'_{k-1} ← new node;       // temp for next call
 9 |  |  u_k ← RelProd(t_k[i], n_2k[i][i'], t_k[i'], r'_{k-1});
⋄10 |  |  if u_k ≠ 0 then add arc (i, i') to A^∃_{t_k};
11 |  |  t_k[i'] ← (t_k[i'] ∪ u_k);          // collect states
*12 |  |  r_k[i'] ← (r_k[i'] ∪ r'_{k-1}); // collect recurring
13 until t_k unchanged;
⋄14 N_SCC ← TransitionsInSCC(A^∃_{t_k});
○15 if DetectSCC(B(r_k), N_≤k, N_SCC) then
16 |  terminate with counterexample;
17 return CheckUnique(t_k);
```

Algorithm 2. Saturate

```
   input    : s_k, n_2k, o_k : node
   // s_k: node to be saturated,
   // n_2k: next state node,
   // o_k: old node
   in-out   : r_k : node
   // r_k: recurring states
   output   : node
 1 if s_k = 1 ∧ n_2k = 1 then
*2 |  if o_k = 1 then
 3 |  |  r_k ← 1;   // recurring state found
 4 |  return 1;
 5 t_k ← new node;
 6 foreach s_k[i] ≠ 0 ∧ n_2k[i][i'] ≠ 0 do
*7 |  r'_{k-1} ← new node;
 8 |  t_k[i'] ← (t_k[i'] ∪
   |  RelProd(s_k[i], n_2k[i][i'], o_k[i'], r'_{k-1});
*9 |  r_k[i'] ← (r_k[i'] ∪ r'_{k-1});
10 t_k ← Saturate(CheckUnique(t_k));
11 return t_k;
```

Algorithm 3. RelProd

formula is declared *invalid* in terms of the system. If saturation finishes and *DetectSCC* never returns *true*, the formula is declared *valid*.

Algorithm 2 and 3 presents the complete algorithm. Lines different from the original saturation algorithm are marked. Although it is crucial to implement, caching is now omitted for the sake of simplicity. *CheckUnique* is used to avoid the duplication of decision diagram nodes. If an equivalent node has already been registered, it returns that node, otherwise registers the input. The decision diagram representation of a next-state function has $2k$ levels. Even levels encode *from* states and odd levels encode *to* states. Custom functions are *TransitionsInSCC* and *DetectSCC*. The former performs an explicit SCC computation (e. g., [15]) on the abstraction and returns transitions of the state space corresponding to abstract arcs in an SCC. The latter is presented in Algorithm 1.

Lines marked with $*$ belong to the computation of recurring states. To identify recurring states, an additional node representing old states is passed to *RelProd*. Reached states that are also in the set of old states are collected similarly to the approach of *constrained saturation* [17]. Sign $\diamond$ marks lines corresponding to explicit search. The node-wise abstraction is built on-the-fly, then *TransitionsInSCC* is used to extract candidate SCCs. Finally, on the line marked with $\circ$, *DetectSCC* is called with the set of recurring states, $\mathcal{N}_{\leq k}$, and transitions in the candidate SCC to perform the incremental fixed-point computation.

7 Related Work

This section briefly summarizes different approaches to SCC computation, from traditional SCC-hull algorithms to SAT-based solutions.

SCC-hull algorithms are usually variants of the algorithm of Emerson and Lei [14]. They solve the SCC computation problem by computing a least fixed

point of the state space that is sure to contain at least one SCC. An SCC-hull is a superset of states belonging to an SCC, thus it proves only the *existence* of an SCC. The incremental SCC computation algorithm presented in Section 4 is also based on the idea of SCC-hull computation. However, our method is tuned to work on the fly, exploiting the results of previous runs to provide incrementality.

Saturation-based SCC computation has also been proposed. The algorithms implemented in [18] are different from SCC-hull algorithms, because both the algorithm of Xie and Beerel and the Transitive Closure method aim to compute exactly those states that belong to an SCC. Because of the caching mechanism of saturation, these algorithms can be very efficient to compute an exact counterexample detected by our algorithm.

An extensive approach to using abstraction in SCC computation has been proposed in [16]. By defining a lattice of abstractions based on one or more components of the model, the paper presents strategies of using some of the abstractions to discard uninteresting parts of the state space and search in relevant components. While node-wise abstraction can be interpreted in that context, the paper uses abstractions similar to the must abstraction presented in Section 5.2 and only accepting states are used to prove the lack of SCCs.

On-the-fly approaches to SCC computation and thus model checking also exist [1, 11–13]. One particularly interesting solution is described in [10]. The paper describes two types of abstractions used to achieve on-the-fly search, also using saturation as a state space generation algorithm.

A different approach in SAT-based model checking is the recent approach called IC3 [2]. By constructing a series of small intermediate lemmas, the k-liveness algorithm [7] identifies one-way walls that separate the possible spaces of SCCs. In this sense, the idea is similar to that of node-wise abstraction.

8 Evaluation

To demonstrate the efficiency of the presented new algorithm (referred to as *Hyb-MC*), models of the Model Checking Contest[4] have been used to compare it to three competitive tools. NuSMV2 [6] is a BDD-based model checker implementing traditional SCC-hull algorithms and is well-established in the industrial and academical community. Its successor, nuXmv [3] implements a k-liveness algorithm [7] based on IC3 for LTL model checking[5]. ITS-LTL is a powerful tool based on saturation that implements various optimizations both for symbolic encoding and on-the-fly SCC detection.

All four tools were run on 7 850 inputs: 27 scalable models of the Model Checking Contest were used to obtain a total of 157 different instances, each checked against 50 randomly generated LTL formulae produced by SPOT [9]. The models represent the behavior of mainly asynchronous, concurrent systems. Out of the successfully checked cases, properties were fulfilled 2 811 times, while 3 565 cases gave negative results. In 1 474 cases, all the tools exceeded the time

[4] `http://mcc.lip6.fr/`

[5] nuXmv was executed with flag "-check_ltlspec_klive".

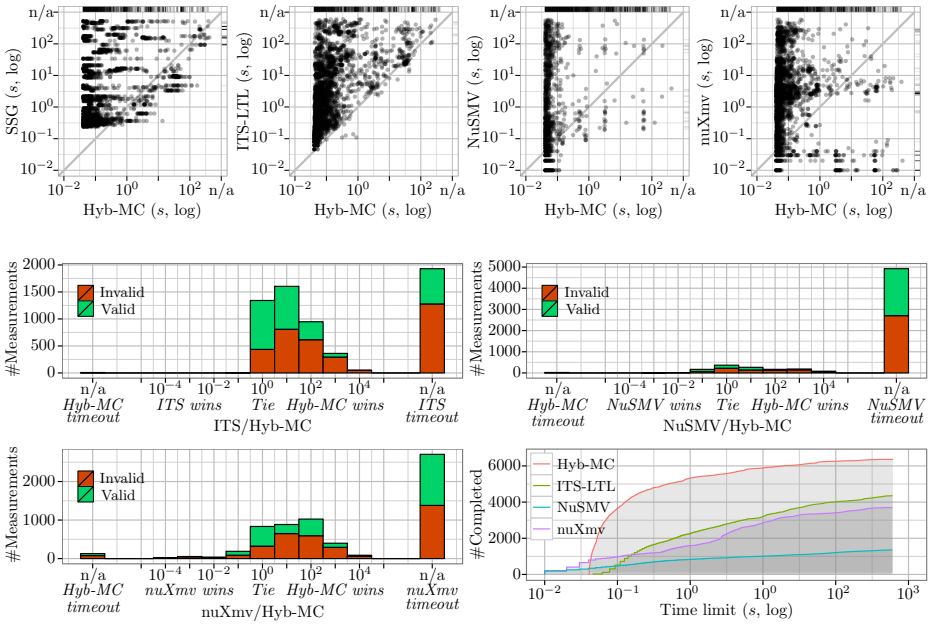


Fig. 4. Measurement results.

limit. Generated expressions contained a nearly equal number of safety and guarantee properties, as well as obligation formulae and more complex properties.

Measurements were done on identical server machines with Intel Xeon processors (4 cores, 2.2GHz) and 8 GB of RAM, with timeout set to 600 seconds. The decision diagram based tools used the same variable ordering produced by heuristics of the ITS toolset. Runtimes were measured internally by every tool, usually including every step of the model checking process (in case of NuSMV and nuXmv, internal transformation of the input was omitted from the result). A prototype of Hyb-MC was implemented in .NET to conduct the measurements.

Results can be seen on Figure 4. On the scatterplots, each point represents a single pair of model instance and property. The runtime of Hyb-MC is always on the x-axis, while the runtimes of state space generation, ITS-LTL, NuSMV and nuXmv are on the y-axis of the subfigures. A point above (below) the diagonal is a measurement where Hyb-MC solved the problem faster (slower). The borders of the diagrams represent the timeout of a measurement for one of the tools. As the plots show, Hyb-MC usually finishes the verification faster than the state space generation of the model, mainly because of on-the-fly operation and efficient incremental operation. State space generation could be finished for some models where model checking was unsuccessful, the overhead of model checking of these complex properties could not be compensated by the incremental operation. Comparing to the other model checking approaches, the vast majority of cases show the competitiveness of our algorithm. The three histograms depict the

differences of runtimes: the bar in the middle shows cases where runtimes of the tools were in the same order of magnitude, while every bar to the left or right means an additional order of magnitude in the runtime of the corresponding tool compared to the other. The last diagram shows the number of cases in which a tool was able to finish the verification within the given time.

Analysis of collected data showed differences in the scalability of the algorithms. While Hyb-MC and ITS-LTL is better in handling a huge number of state variables, NuSMV and nuXmv performed much better on models with state variables of large domains. Only nuXmv's k-liveness algorithm proved to be sensitive to different classes of properties, the other tools did not show significant differences in the distribution of runtimes. During the measurements, Hyb-MC spent only 17% of the time computing SCCs. Overall, 359 084 symbolic fixed point computations were started, while abstraction and explicit algorithms prevented $1.22 \cdot 10^8$ runs of symbolic SCC computation, 99.7% of all the cases. 89% of these cases were prevented by the absence of recurring states (as a first check), while the remaining 11% were the cases where explicit runs on node-wise abstractions managed to find even more evidence.[6]

9 Conclusion and Future Work

In this paper, a new algorithm has been presented for LTL model checking. The described approach divides model checking into smaller tasks, and handles large state spaces by performing efficient local computations on the components. The absence of SCCs is proved with the help of a specialized abstraction function and inductive reasoning, while existing SCCs are discovered by a new incremental symbolic fixed point algorithm. These solutions constitute an efficient on-the-fly, hybrid model checking approach that combines the advantages of explicit and symbolic algorithms. Our solution uses saturation for state space traversal, which makes it suitable for concurrent systems. Extensive measurements justified this claim for the models of the Model Checking Contest.

The presented algorithm has a huge potential for future development. Following the idea of driving the symbolic algorithm with explicit runs, a promising direction is to combine partial order reduction with symbolic model checking. In addition, we also plan to use advanced representations of the properties to further improve the speed of model checking.

References

1. Biere, A., Zhu, Y., Clarke, E.: Multiple state and single state tableaux for combining local and global model checking. In: Correct System Design, LNCS, vol. 1710, pp. 163–179. Springer (1999)
2. Bradley, A.: Understanding IC3. In: Theory and Applications of Satisfiability Testing – SAT 2012, pp. 1–14. No. 7317 in LNCS, Springer (2012)

[6] For a detailed analysis of collected data, cf. `http://inf.mit.bme.hu/en/tacas15`.

3. Cavada, R., Cimatti, A., Dorigatti, M., Mariotti, A., Micheli, A., Mover, S., Griggio, A., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. Tech. rep., Fondazione Bruno Kessler (2014)
4. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: an efficient iteration strategy for symbolic state space generation. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2031, pp. 328–342. Springer (2001)
5. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. on Softw. Tools for Technology Transfer 8(1), 4–25 (2006)
6. Cimatti, A., Clarke, E., Giunchiglia, E., et al.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification, LNCS, vol. 2404, pp. 359–364. Springer (2002)
7. Claessen, K., Sorensson, N.: A liveness checking algorithm that counts. In: Formal Methods in Computer-Aided Design, 2012. pp. 52–59. IEEE (2012)
8. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Computer-Aided Verification, LNCS, vol. 531, pp. 233–242. Springer (1991)
9. Duret-Lutz, A., Poitrenaud, D.: SPOT: An extensible model checking library using transition-based generalized Büchi automata. In: Proc. of the IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. pp. 76–83 (2004)
10. Duret-Lutz, A., Klai, K., Poitrenaud, D., Thierry-Mieg, Y.: Combining explicit and symbolic approaches for better on-the-fly LTL model checking. arXiv:1106.5700 [cs] (2011)
11. Haddad, S., Ilié, J.M., Klai, K.: Design and evaluation of a symbolic and abstraction-based model checker. In: Automated Technology for Verification and Analysis, LNCS, vol. 3299, pp. 196–210. Springer (2004)
12. Klai, K., Poitrenaud, D.: MC-SOG: An LTL model checker based on symbolic observation graphs. In: Applications and Theory of Petri Nets, LNCS, vol. 5062, pp. 288–306. Springer (2008)
13. Sebastiani, R., Tonetta, S., Vardi, M.: Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking. In: Computer Aided Verification, LNCS, vol. 3576, pp. 350–363. Springer (2005)
14. Somenzi, F., Ravi, K., Bloem, R.: Analysis of symbolic SCC hull algorithms. In: Formal Methods in Computer-Aided Design, LNCS, vol. 2517, pp. 88–105. Springer (2002)
15. Tarjan, R.: Depth first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
16. Wang, C., Bloem, R., Hachtel, G.D., Ravi, K., Somenzi, F.: Compositional SCC analysis for language emptiness. Form. Method. Syst. Des. 28(1), 5–36 (2006)
17. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Automated Technology for Verification and Analysis, LNCS, vol. 5799, pp. 368–381. Springer (2009)
18. Zhao, Y., Ciardo, G.: Symbolic computation of strongly connected components and fair cycles using saturation. Innov. Syst. Softw. Eng. 7(2), 141–150 (2011)

Acknowledgements. This work was partially supported by the ARTEMIS JU and the Hungarian Research and Technological Innovation Fund in the frame of the R5-COP project.