Parallel Saturation Based Model Checking

András Vörös*, Tamás Szabó, Attila Jámbor, Dániel Darvas, Ákos Horváth, [+]Tamás Bartha

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
*vori@mit.bme.hu

[+]Computer and Automation Research Institute
MTA SZTAKI
Budapest, Hungary

Abstract— Formal verification is becoming a fundamental step of safety-critical and model-based software development. As part of the verification process, model checking is one of the current advanced techniques to analyze the behavior of a system. In this paper, we examine an existing parallel model checking algorithm and we propose improvements to eliminate some computational bottlenecks. Our measurements show that the resulting new algorithm has better scalability and performance than both the former parallel approach and the sequential algorithm.

Keywords: model checking, parallel, saturation, Petri Net, state space

I. INTRODUCTION

Formal methods are widely used for the verification of safety critical and embedded systems. The main advantage of formal methods compared to extensive testing is that either they can provide a proof for the correct behavior of the system, or they can prove that the system does not comply with its specification. On the contrary, testing can only examine a portion of the possible behaviors.

One of the most prevalent techniques in the field of formal verification is *model checking*, an automatic technique to check whether a system fulfills specification. Model checking needs a representation of the state space in order to perform analysis. Generating and storing the state space representation can be difficult in cases where the state space is very large.

There are two main problems causing the state space to explode:

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

Symbolic methods [6] are an advanced technique to handle state space explosion. Instead of storing states explicitly, symbolic techniques rely on an encoded representation of the state space such as decision diagrams. These are compact graph representations of discrete functions. *Saturation* [4][7][17] is considered as one of the most effective model checking algorithm, which combines the efficiency of symbolic methods with a special iteration strategy.

Time efficiency is also critical in model checking. As symbolic methods solved many of the memory problems, the demand to develop faster model checking algorithms increased. In the current paper, we choose to utilize the computational power of recent multi-core processors or multi-processor architectures. Our work focuses on developing a parallel model checking algorithm, based on a former parallel saturation model checking algorithm published in [5].

The remainder of the paper is structured as follows: sect. II introduces the background of our work, the modeling formalism: Petri Nets, decision diagrams and model checking. The basic parallel saturation algorithm is presented in sect. III. In sect. IV we present our work and improvements. In sect. V we provide our measurements, while Sect. VI summarizes the related work. Our future plans are found in the last section.

II. BACKGROUND

Petri nets [1] are graphical models for concurrent and asynchronous systems, providing both structural and dynamical analysis. A (marked) discrete Petri net is a 5-tuple: $N = (P, T, w^-, w^+, M_0)$ represented graphically by a digraph, where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $P \cap T = \emptyset$, $w(p,t)$: $P \times T \rightarrow N$ is the input, $w(t,p)$: $T \times P \rightarrow N$ is the output incidence function for each transition, represented by weighted arcs from places to transitions and from transitions to places; $M_0 : P \rightarrow N$ is the initial marking, represented by $M(p_i)$ tokens in place p_i for every i. A transition is enabled, if for every incoming arc of t: $M(p_i) \geq w(p_i,t)$. An *event* in the system is the firing of an enabled transition t_i, which decreases the number of tokens in the incoming places with $w(p,t_i)$ and increases the number of tokens in the output places with $w(t_i,p)$. The state space of the Petri Net is the set of states reachable through transition firings. Figure 1. depicts an example Petri Net model of a producer-consumer system using a buffer (capacity is 1) for synchronization. There are many ways to store the set of reachable states of a Petri Net. In our work, we used decision diagrams for it.

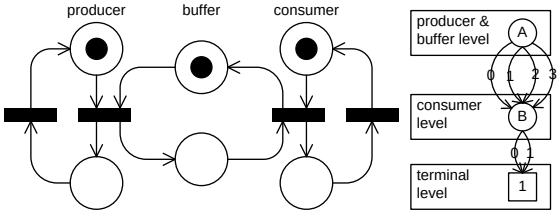

Figure 1. Petri Net model and its state space representation

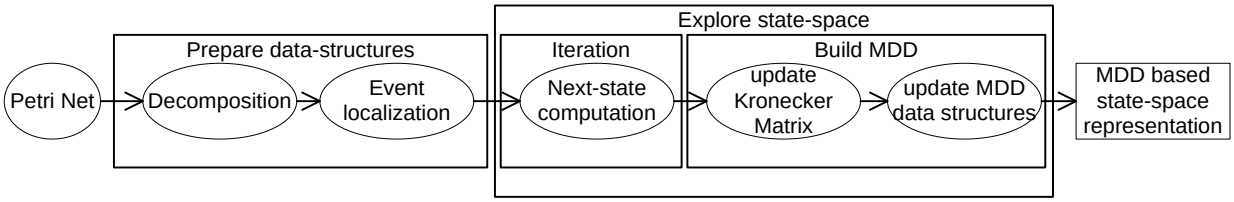

Figure 2. Overview of the saturation algorithm

A *Multiple Valued Decision Diagram* (MDD) [2][3] is a directed acyclic graph, with a node set containing two types of nodes: non-terminal and two terminal nodes. The nodes are ordered into levels. A non-terminal node is labeled by a variable index *k*, which indicates to which level the node belongs (which variable it represents), and has n_k (domain size of the variable, in binary case n_k=2) arcs pointing to nodes in level *k-1*. Duplicate nodes are not allowed, so if two nodes have identical successors in level *k*, they are also identical. Redundant nodes are allowed: it is possible that a node's all arcs point to the same successor. These rules ensure that MDD-s are canonical representation of a given function or set. Figure 1. depicts beside the example Petri Net its MDD representation. It encodes the state space, which contains 8 states. These are encoded in the paths from root (*A*) to the terminal one.

Traditional *symbolic model checking* [6] uses encoding for the traversed state space, and stores this compact encoded representation only. Decision diagrams proved to be an efficient storage, as applied reduction rules provide a compact representation form. Another important advantage is that symbolic methods enable us to manipulate large set of states efficiently.

The first step of symbolic state space generation is to encode the possible states. Traditional approach encodes each state with a certain variable assignment of state variables $(v_1, v_2 \dots v_n)$, and stores it in a decision diagram. To encode the possible state changes, we have to encode the transition relation, the so called *Next-state* function. This can be done in a 2*n* level decision diagram with variables: $\mathcal{N} = (v_1, v_2 \dots v_n, v_1', v_2' \dots v_n')$, where the first *n* variables represent the "*from"*, and second *n* variables the "*to"* states. The *Next-state* function represents the possibly reachable states in one step. Usually the state space traversal builds the *Next-state* relation during a breadth first search. The reachable set of states $\mathcal{S}$ from a given initial state s_g is the transitive closure (in other words: the fixed-point) of the *Next-state* relation: $\mathcal{S} = \mathcal{N}^*(s_g)$.

Saturation based state space exploration [4][7] differs from traditional methods as it combines symbolic methods with a special iteration strategy. This strategy is proved to be very efficient for asynchronous systems modeled with Petri Nets. The saturation algorithm consists of the following steps depicted in Figure 2.

1) Decomposition: Petri Nets can be decomposed into local sub-models. The global state can be represented as the composition of the components' local states: $s_g = (s_1, s_2, \dots, s_n)$, where *n* is the number of components. This decomposition is the first step of the saturation algorithm. Saturation needs the so called *Kronecker consistent* decomposition [4][16], which means that the global transition (*Next-state*) relation is the Cartesian product of the local-state transition relations. Formally: if $\mathcal{N}_{i,e}$ is the *Next-state* function of the transition (*event*) *e* in the *i*-th sub-model, the global *Next-state* of event *e* is: $\mathcal{N}_e = \mathcal{N}_{1,e} \times \mathcal{N}_{2,e} \times \dots \times \mathcal{N}_{n,e}$. In case of asynchronous systems, a transition usually affects only some or some parts of the sub-models. This kind of event locality can be easily exploited with this decomposition. Petri nets are Kronecker consistent for all decompositions.

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

3) Special iteration strategy: Saturation iterates through the MDD nodes and generates the whole state space representation using a node to node transitive closure. In this way saturation avoids that the peak size of the MDD to be much larger than the final size, which is a critical problem in traditional approaches [7]. Let $\mathcal{B}\langle k, p\rangle$ represent the set of states represented by the MDD rooted at node *p*, at level *k*. Saturation applies $\mathcal{N}^*$ locally to the nodes from the bottom of the MDD to the top. Let ε be the set of events affecting the *k*-th level and below, so $top_e \leq k$. We call a node *p* at level *k* saturated, if node $\mathcal{B}\langle k, p\rangle = \cup_{\forall e \in \varepsilon} \mathcal{N}_e^*(\mathcal{B}\langle k, p\rangle)$. The state space generation ends when the node at the top level becomes saturated, so it represents $\mathcal{S} = \mathcal{N}^*(s_g)$.

4) Encoding of the Next-state function: The formerly presented Kronecker consistent decomposition leads to sub-models, where the *Next-state* function can be expressed locally, with the help of the so called Kronecker matrix [8]. This is a binary matrix, that contains 1 at level *k*, iff: $\mathcal{N}_{k,e}[i, j] = 1 \leftrightarrow j = \mathcal{N}_{k,e}(i)$. It represents only the local next states. This representation turned out to be very efficient in practice [7].

5) Building the MDD representation of the state space: At first we build the MDD representing the initial state. Then we start to saturate the nodes in the first level by trying to fire all events where $top_e = 1$. After finishing the first level, we saturate all nodes at the second level by firing all events, where $top_e = 2$. If new nodes are created at the first level by the firing, they are also saturated recursively. It is continued at every level *k* for events, where $top_e = k$. When new nodes are created in a level below the current one, they are also recursively saturated. If the root node at the top level is

saturated, the algorithm terminates. Now, the MDD represents the whole state space with the next state relation encoded in Kronecker matrices.

6) State space representation as an MDD: A level of the MDD generated during saturation represents the local state space of a submodel. The possible states of the sub-model constitute the domain of the variables in the MDD, so each local state space is encoded in a variable.

Saturation is a hard problem from the parallelization point of view [5]. Since the iteration computes local fixed-points, it has to compute the union of sets at every node, which should be synchronized in order to avoid inconsistent and redundant operations. In addition, the algorithm uses caching mechanisms at every level for union operations, node storage and next state computations, which means additional synchronization overhead.

III. Overview of the Parallel Algorithm

In this section we introduce the algorithm presented in [5]. This algorithm served as the basis of our improved algorithm, which is presented in section 4.

The authors of [5] divided the saturation into several stages, and assigned the computation of a node to a thread. Node computations and operations consist of:

- node management in the MDD data structures,
- event and next state computations,
- node modifications,
- the manipulation of the MDD rooted at this node by recursive calls.

These tasks are executed either by one thread, or this thread calls another one to do them. The logic of which tasks are outsourced by a thread to another is a critical point. These tasks should be large enough to avoid increase in synchronization and communication overhead, but they also should be reasonable size to enable more threads to work parallel. The main aim is to avoid inconsistent MDD states, as this will prevent the algorithm to reach the fixed-point. It is ensured by the proper synchronization and locking mechanism.

Synchronization of data structures: The algorithm uses decision diagrams, therefore it has to take care of the consistency of their underlying hash tables (so called unique table [2]). The motivation was to enable as many threads to manipulate nodes simultaneously as many possible. The algorithm synchronizes at every level, in this way it avoids inconsistent MDD levels. The responsibility for global MDD consistency is left to the iteration, which is preserved with locking sub-MDDs when they are manipulated.

Synchronization of MDD operations: The presented algorithm uses a special locking strategy to preserve MDD consistency. As MDD serves as the underlying data structure for the iteration, it is important from the saturation point of view. A classical decision diagram approach was used in [5], so at every operation the argument MDD-s are locked in order to prevent concurrent manipulation. This means relatively high synchronization overhead but it is essential. However, saturation tries to avoid operations on the whole decision diagram, instead it computes local operations. This means locking only sub-MDDs, so the algorithm itself ensures smaller locking overhead. Therefore small MDD operations are a characteristic of saturation.

Synchronization of the iteration: The iteration order is also important. The threads have to synchronize the operations executed on nodes. The locking strategy is simple: one thread can access a node and locks it. During the next state iteration, the sub-MDD rooted in it is also locked. It is clear that the algorithm can run parallel only in the case when more nodes appear in the levels, so the MDD is getting wider.

The iteration is synchronized with the help of node arguments. Every node has a counter for the tasks which are under execution or are planned to be executed. This counter prevents the algorithm to miss operations so that it can avoid unfinished operation sequences.

In order to preserve dependencies, the algorithm introduces upward arcs (Figure 3.). These arcs represent dependencies in the iteration order, so if a node has an upward arc pointing to an upper node means: a thread computed the firing at the upper node and it called another thread to compute the lower levels of the MDD rooted there.

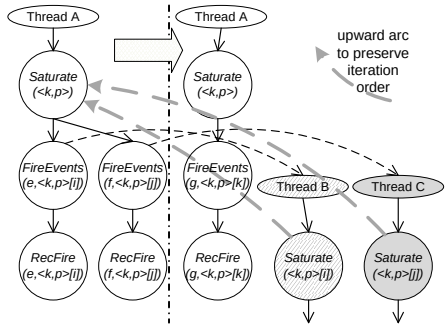

Figure 3. Parallel computations in saturation

The algorithm also avoids redundant computations by cache synchronization: when a thread starts computing a part of the reachable state space, it signs it in the cache with the value of the actually processed node. This way, if another thread would start exploring that part of the state space, it easily realizes that it is still being processed, so it avoids redundant exploration and just registers itself for the result.

The formerly introduced fixed-point computations are calculated parallel in this algorithm. In the former section we showed the synchronization and locking mechanisms, here we give an insight to the main operation of the algorithm. A node p at level k is signed $\langle k,p\rangle$, and the i-th arc of this node is: $\langle k,p\rangle[i]$. Functions for the fixed-point computations are the following:

- $\mathcal{N}^*_{\leq k}(\mathcal{B}\langle k,p\rangle)$ is the full state space represented by $\mathcal{B}\langle k,p\rangle$, it is computed by function *Saturate*$(\mathcal{B}\langle k,p\rangle)$
- $\mathcal{N}^*_e(\mathcal{N}^*_{<k}(\mathcal{B}\langle k,p\rangle[i]))$ is the transitive closure of the application of a next state function restricted to event *e*, this is computed by *FireEvents*$(\mathcal{B}\langle k,p\rangle[i])$
- $\mathcal{N}^*_{\leq k}(\mathcal{N}_e(\mathcal{B}\langle k,p\rangle))$ is the transitive closure of the state space reached through an event *e* at level *k*. This is computed by function *RecFire*$(e, \mathcal{B}\langle k,p\rangle)$.

- reaching $\mathcal{N}^*_{\leq k}(\mathcal{B}\langle k,p\rangle)$ can be only ensured, if when the computations below $\langle k,p\rangle$ are finished, the algorithm continues saturating $\langle k,p\rangle$. The function is called *NodeSaturated*($\langle l,q\rangle$). It is called when at level l = *k-1* node q is saturated, and it continues computing the transitive closure at node $\langle k,p\rangle$. This way the algorithm ensures consistent saturated end state.

The operation of the work distribution in the algorithm is depicted in Figure 3. In this figure a thread (*Thread A*) starts saturating a node ($\langle k,p\rangle$). During the computation some recursive calls are needed. These calls are outsourced to other threads. In order to preserve the iteration order, these threads set an upward arc to the upper node ($\langle k,p\rangle$). This way the upper node could not be finished until the nodes below are finished.

In addition to the above defined functions, the parallel algorithm published in [5] uses *Remove*($\langle k,p\rangle$) function for removing dead endings from the MDD. These are created when a parallel thread starts a computation of a firing of a dead transition, which cannot fire from the given marking. Functions, which are responsible for synchronization: *Lock*($\langle l,q\rangle.dw$) and *Unlock*($\langle l,q\rangle.dw$). These functions lock the MDD data structure downward (Figure 4.) in order to prevent concurrent manipulation.

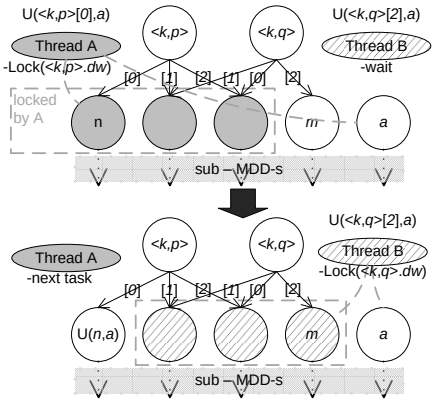

Figure 4. Locking mechanism

The locking ensures that the iteration order is preserved, and operations executed on nodes are not interfered by each other. The algorithm is proved to be correct [5], as it ensures:

- correct iteration order, by removing synchronization methods we get the sequential algorithm
- correct synchronization of the data structures, both in the MDD operations, and both in the next state representations
- since locks ensure that updating a node is atomic, firing transitions exhaustively will result in the same MDD shape for a saturated node as in the sequential algorithm

A. Difficulties in the parallelization

Parallel implementation of saturation involves a big synchronization overhead, making efficient parallelization difficult. This also emphasizes the fundamental role that the proper synchronization plays in parallel realization of the saturation algorithm. There are two main bottlenecks: first is that parallelization of state space exploration is generally a hard task. In order to avoid redundant state exploration, we have to ensure that the parallel directions synchronize properly without dramatically increasing the synchronization costs. Another reason is that saturation uses a special underlying data structure: decision diagrams. Parallelizing decision diagram operations involves a big synchronization overhead, caused by the fact that decision diagrams are built in a bottom-up fashion, where upper levels highly depend on lower levels. As measurements showed in [5], the parallel saturation algorithm runs faster on more processors than on one, but still remains slower than the sequential algorithm by 10-300%. Scalability is also an important factor is parallelization. By scalability we mean the following two characteristics:

- The runtime of the algorithm will decrease with respect to the increasing number of resources.
- The relative speed of the parallel algorithm will increase comparing to its sequential counterpart with the growing number of tasks

It is important to examine the scalability of the parallel algorithm. Experiments [5] showed that independent on how much the resources were increased; the parallel algorithm could not exceed the speed of the sequential one. In addition, for most models the parallel algorithm could not exploit the increasing number of tasks meant by bigger models, for most cases the handicap of the parallel algorithm remained for big models as well.

IV. Details of our new Algorithm

We have developed a new synchronization mechanism to improve the algorithm presented in [5]. Our aim was to localize the effect of the locks and to reduce the overhead caused by them. Our improvements led to significant speed-up of the algorithm. We introduce local synchronization, which avoids downward locking (i.e. *Lock*($\langle l,q\rangle.dw$) and *Unlock*($\langle l,q\rangle.dw$) calls). The problem with downward locking is not only its overhead. In many cases, the inefficient synchronization makes the threads unable to run parallel, even when it would not be necessary for them to wait for each other.

A. Main features

We have developed a new synchronization method instead of downward locking. We use a flag in the node data structure to enable threads locking nodes. With the help of this flag we could use atomic operations on nodes, without making the MDD operations mutually exclusive. This locking mechanism is applied in the fixed point computation at every iteration step, when the set represented by the node is augmented. As functions *Recfire*, *FireEvents* and *NodeSaturated* are all augmenting the set represented by the node, they all use this new synchronization strategy. Our strategy is depicted in Figure 5.

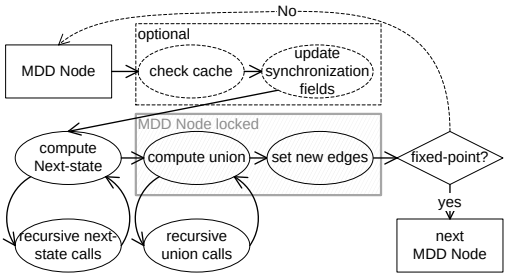

Figure 5. Workflow of our parallel saturation algorithm

Our locking strategy is proved to be the proper solution, because of two reasons:

- if the validity of the locking would be reduced, we could lose information, the algorithm would be slower,
- making stricter the locking strategy would increase overhead without any advantage.

The first case can be easily demonstrated with the following example. If we lock the node only for the setting of a new edge, omitting the union from it, it can happen that two threads compute parallel for edge $\langle k,p\rangle[i] = a$ a new value: $a' = a \cup b$ and $a'' = a \cup c$, and substitutes it with it. Letting the threads run parallel may result $\langle k,p\rangle[i] = a'$ or $\langle k,p\rangle[i] = a''$, instead of $a''' = a \cup b \cup c$. When the algorithm realizes that the fixed point is still not reached, corrects the edge $\langle k,p\rangle[i]$, but it means that some of the former computations are unnecessary.

We have to serialize the computations of $\langle k,p\rangle = \langle k,p\rangle \cup Next\ state(\langle k,p\rangle)$ in order to avoid losing information. By locking nodes during union computation the algorithm preserves the iteration order, meanwhile increased parallelism is reached by restricting the scope of the locks.

B. Implementation

We have developed a complex synchronization mechanism in the data structure level of saturation to prevent data races and to ensure consistent execution.

We have implemented a mutually exclusive access to the data structures of the *Next-state* computation, such as Kronecker matrices and globally reachable states, which contains the mapping from the Petri Net states to the MDD variables' domains. The MDD data structures are serialized at every level, in this way we can preserve the consistency of the algorithm. The MDD operations used during the building of the MDD do not need additional synchronization; it uses the same MDD level locking mechanism for the modifications. We could avoid additional operation synchronization with the use of constructive operations, so that the union operation does not consume its arguments, but creates a new MDD representing the union instead. This may lead to great number of unnecessary nodes, which should be cleaned from the data structures. Every node has a counter counting the references pointing to it, so that we can decide at any time to clean the data structures and we can easily decide which node is necessary and which is not. The algorithm introduced in [5] presented a pre-cache mechanism to avoid redundant state space exploration. We have implemented this method in our approach too. Using this cache for synchronization helps avoid redundant state-space computations. We only have to register the event and the node immediately if the event is executed on it (*RecFire*). All other threads intending to explore the same sub-state space will realize that it is being now executed, and the new threads just register themselves for the result. The synchronization of this cache is important. We do not use a global cache; instead we assign a cache to each level. This reduces the synchronization costs. The same strategy is used for the union operation, as the algorithm does not lock the operations, just the MDD levels and caches for the time of modifications. This strategy enables the parallel computation of $a = b \cup c$ and $e = c \cup d$, which was a shortcoming in former algorithms. This leads to increased parallelism and reduced overhead.

C. Correctness of the algorithm

The correctness of saturation was proved in many papers, we refer the reader to [4][7]. The basic parallel saturation algorithm was presented in [5], where the correctness of the algorithm is also proved. The main problem with parallel saturation is if the iteration order is corrupted, then the final result is just the subset of the real state space. In order to avoid it, the sequential algorithm was completed with the locking and the proper work distribution mechanisms. These modifications let the algorithm run hardly parallel, which is confirmed by the measurements in [5].

Our modifications enable the algorithm to exploit the resources of recent multiprocessor architectures more efficiently, and we prove the correctness of our approach. In this paper we discuss only modifications affecting the iteration order, as other improvements are implementational.

The modified algorithm should:

- Preserve iteration order,
- Reach saturated final state,
- Preserve consistency of data structures.

Iteration order is not affected by our modifications, so we refer the reader to [5] for a complete proof. We used the same functions for the computation of $\mathcal{N}^*_{\leq k}(B\langle k,p\rangle)$, $\mathcal{N}^*_e(\mathcal{N}^*_{<k}(B\langle k,p\rangle[i]))$ and $\mathcal{N}^*_{\leq k}(\mathcal{N}_e(B\langle k,p\rangle))$. Consequently calling these functions preserves the iteration order. In addition, after an iteration is finished calling function *NodeSaturated* ensures that every node encodes $\mathcal{N}^*_{\leq k}(B\langle k,p\rangle$, so the iteration is complete. Our modifications may change the order of union functions. However, as union is commutative, this doesn't change the final result. These all ensures reaching a saturated final state. Note that the consistency comes from that the function *NodeSaturated* is called every time when the computation of a node is finished. It finalizes the nodes in the data structures.

The last condition is highly affected by our new locking strategy. Consistent data manipulation is required to ensure global consistency. It is important to examine whether this condition holds. Our approach omits downward locking and preserves consistency without locking the arguments of the union operation. From the consistency point of view the most important condition is to assure that the arguments of

the operations are finalized. The most important assumption is that the algorithm performs MDD operations only on nodes, which are permanently in the MDD data structures, so they will not change any more. This assumption is proved by induction. At the beginning of the algorithm, all edges are set to terminal nodes, so the condition holds. After this, the algorithm sets upward-arcs until a node becomes saturated, so no union is called on temporary nodes. When first a node becomes saturated, it is placed into the MDD data structure, so it is finalized. This point is when the algorithm executes union operation. Now, one argument of the union is the newly saturated node, and the other argument is the old edge, which points either to terminal one or terminal zero (by default). Both are permanently in the data structure. So the operation can be executed and has a consistent result.

From now, we call union in two cases. When a node is saturated, we call union in function *NodeSaturated*. In this case the algorithm computes union of a recently saturated node with the old edge of the upper node, which is saturated. Both nodes are finalized, the result is consistent.

The other case is when computation requires a node from the cache. The algorithm uses the value only in the case if it is saturated. So this argument of the operation is finalized. The other one is also saturated as it was formerly the endpoint of a node's arc, the result will be also consistent.

V. Evaluation of the Algorithm

A. Environment

We have developed an experimental implementation in the Microsoft C# programming language. We used some of the framework's built-in services, like *ThreadPool* and locking mechanisms. We examine our algorithm and compare our approach both to a sequential algorithm written in C#, and to the implementation written in *C* programming language [5][1]. We used a desktop PC for the measurements: *Intel Core2 Quad CPU Q8400 2,66GHz, 4 GB memory*. For our implementation we used *Windows 7 Enterprise, .NET 4.0 x64*. For the implementation from [5] we used *Ubuntu 10.10 with gcc-4.4*. Comparing the performance of [5] and our approach is a little bit difficult. Our approach computes the local states dynamically. In contrast the algorithm [5] needs a pre-computation step, and works with a formerly computed Kronecker representation, so they are two different variants of saturation. Former measures [4] showed that with the use of precomputed Kronecker representation 50-60% speed up can be gained. However in most cases the user has to adapt the model to some special requirements [4], so it is more difficult to use. The models we used for the evaluation are widely known in the model checking community. *Flexible Manufacturing System* (*FMS*) and *Kanban* system are models of production systems [7]. The parameter *N* refers to the complexity of the model and it influences the number of the tokens in it. *Slotted Ring* (*SR*) and *Round Robin* (*RR*) are models of communication protocols [4], where *N* is the number of participants in the communication. The state spaces of the models range from 10^{15} up to 10^{150}.

B. Runtime and speed-up results

TABLE I. Runtime results of our algorithm

SR (*N*)	**30**	**60**	**90**	**120**	**150**
sequential	0.66s	4.5s	14.8s	34.7s	70.7s
parallel	0.64s	4.5s	14.4s	33.8s	65.2s
speed-up	1.03	1.0	1.027	1.027	1.084
Kanban (*N*)	**50**	**100**	**200**	**300**	**400**
sequential	0.5s	5.1s	63.2s	295s	890s
parallel	0.4s	2.6s	20.5s	80.6s	228s
speed-up	1.25	1.96	3.08	3.66	3.90
FMS (*N*)	**50**	**100**	**150**	**200**	**250**
sequential	1.7s	14s	61s	180s	444s
parallel	1.2s	7.9s	27.1s	67s	143s
speed-up	1,41	1,77	2,25	2,68	3,10

Slotted Ring: The regular characteristic of the model suggests that it cannot be parallelized well. Our measurements show that the parallel algorithm has the same performance as the sequential one. In addition, as the size of the model grows, the parallel algorithm outperforms the sequential one up to 8.4%. If we compare this result with the former implementation (TABLE II.), the version written in C is faster. It comes from the difference in the programming environment, and also from the fact that the C version uses precomputed Kronecker representation, whose computation time is not included in these measures.

TABLE II. Runtime results of [5], SR model

SR (*N*)	**30**	**60**	**90**	**120**	**150**
sequential	0.2s	1.4s	4.4s	10.2s	19.7s
parallel	0.4s	2.3s	7.5s	17.1s	34.4s
speed-up	0.5	0.61	0.59	0.6	0.57

If we take into consideration only the relative speed of the algorithms, our approach reached 8% runtime gain comparing to its sequential counterpart, while the old one from [5] just about 40% runtime penalty.

Kanban: The state space exploration of the Kanban system was 25% faster with the parallel algorithm for still small models. However, for bigger models the performance gain of the parallel algorithm increased. Last measurement of the parallel algorithm is nearly 4 times as fast as the sequential (TABLE I.). The sequential one from [5] is slower about 1000% than our sequential one (because of the precomputed Kronecker decomposition is not efficient for this model), so the comparison is difficult. However, despite our speed up factor, the parallel algorithm from [5] is about 50% slower than its sequential counterpart.

Flexible Manufacturing System: This model contains also little regularity, this way the parallel algorithm runs at least 41% faster than the sequential one. For large models the sequential algorithm needs 3 times as much time as the

[1] We, the authors would like to thank Dr Jonathan Ezekiel for providing us their program and for all his help.

parallel one. We could not compare this result with [5] because segmentation fault occurred at all the time.

The efficiency of symbolic methods is highly model-dependent. This is especially true for saturation and parallel saturation. Those models that are not appropriate for saturation based verification, could also not be verified with parallel saturation. As parallel saturation usually uses 10-50% more memory than the sequential one, the models which do not fit into memory in the sequential case will also not fit in the parallel case.

In addition, for highly regular models, where sequential saturation turned out to be extremely efficient, parallelization leads to 30-50-100% runtime overhead. These models usually have only few nodes at each level, so they provide less work for parallel threads. Moreover, saturation usually finishes state space generation within a second, so the overhead of creating threads also makes worse the performance of the parallel algorithm. In the following table (TABLE III.) we show the runtime results for two extremely big, but extremely regular models [4]. We present here the measures for our algorithm, but the former program [5] produced similar results in general.

TABLE III. RUNTIME OF NOT PARALLELIZABLE MODELS

model	Dining philosophers	Round Robin
size	1000	1000
sequential	0,91s	17,9s
parallel	1,35s	34,6s

C. Scalability

With this new locking strategy we examined the scaling of the runtimes with the number of used processors. The scalability of the parallel algorithm is also good for most models. Due to the lack of space, we examine here only the results of the FMS model, N = 200. We compare in the following diagram (Figure 6.) the runtime of the sequential algorithm to the parallel one executed on 1-2-3-4 CPU-s. It can be seen that the algorithm scales well with the growing number of processors. In addition, the algorithm is faster than the sequential one still on one CPU. It can efficiently exploit that we can run multiple threads in one CPU.

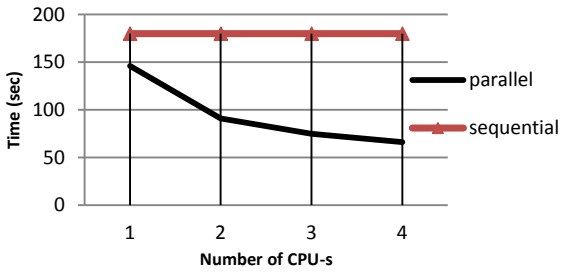

Figure 6. Scaling of the parallel algorithm

We also examined the scaling of the runtimes with the growing size of the models. Figure 7. depicts the runtimes of the parallel and sequential state space generator algorithms presented in this paper for the FMS model. It is easy to see that the advantage of the parallel algorithm grows with the growing number of tasks meant by the bigger models (N is the size of the model).

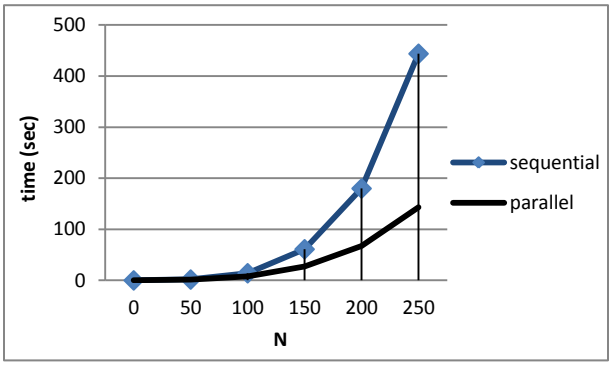

Figure 7. Runtimes of our implementations, FMS model

D. Summary

Our parallel algorithm is more efficient than its sequential counterpart, if we take the runtimes into account. However, from the memory consumption point of view, the situation is different: as parallel threads starts computing more "dead endings" (directions where no solution can be found), memory consumption is usually 10-50% more than for the sequential algorithm.

Comparison of our approach and the former one is quite difficult: as they use neither the same kind of saturation algorithm, nor the same programming environment, runtime results are not easily comparable. However, the speed up factor compared to the algorithms' sequential counterparts suggests that our synchronization strategy leads to the more efficient parallelization of the computation.

VI. RELATED WORK

With the rising number of multiprocessor systems and multi-core processors, several efforts have been made to utilize them in formal verification. To overcome the limitations of model checking, many approaches appeared to investigate the possibilities of parallelization or distribution of the computational work. The most prevalent parallel model checking algorithms are based on explicit enumeration of the states. As their data structures are less complex than symbolic data structures, it causes small computational overhead to synchronize. The PREACH (Parallel Reachability) [10] tool also uses explicit techniques to store and explore the state space of the models. It provides distributed and parallel model checking capabilities. The goal of this project was to develop a reliable, easy to maintain, scalable model checker that was compatible with the Murphi specification language. The program uses the DEMC (Distributed Explicit-state Model Checking) [11] algorithm, it partitions the states with a hash function and assigns each partition to a workstation. Every workstation examines only the states assigned to it. It was showed that this approach can handle large models consisting of nearly 30 billion states.

DiVinE [9] is a tool for parallel shared-memory explicit LTL model-checking and reachability analysis. The tool is based on distributed-memory algorithms re-implemented specifically for multi-core and multi-processor environments using shared memory.

Efficient parallelization of symbolic algorithms is a hard task. Distributing the computational work may increase synchronization cost significantly because of the complex data structures. However, in [12] authors presented a novel distributed, symbolic algorithm for reachability analysis that can effectively exploit a large number of machines working in parallel. The novelty of the algorithm is its dynamic allocation and reallocation of processes to tasks and its mechanism for recovery, from local state explosion. As a result, the algorithm is work-efficient. In addition, its high adaptability makes it suitable for exploiting the resources of very large and heterogeneous distributed, non-dedicated environments. Thus, it has the potential of verifying very large systems.

In [13] authors present a scalable method for parallel symbolic on-the-fly model checking in a distributed memory environment. Their method combines on-the-fly model checking for safety properties with scalable reachability analysis. Their approach has the ability to generate counterexample, where extra memory requirement is evenly distributed among the processes by a memory balancing procedure.

Our aim was to improve saturation based model checking; however other researches were also done in this area. In [5], which served as the basis of our work, authors presented a parallel saturation based model checking algorithm, which we improved significantly in our work. In [14] authors presented a distributed saturation algorithm, which can efficiently exploit the increased memory of network of workstations (NOW). In this way the algorithm could cope with bigger models. Despite our work, this algorithm did not parallelize the algorithm itself.

VII. Conclusion and Future Work

In this paper we presented an improved synchronization method for the parallelization of saturation based model checking. Our improvements led to increased parallelization and performance gain comparing to former approaches. However, the parallelization is highly dependent on the structure of models. Saturation is extremely efficient for some models. The parallelization of these models cannot lead to further speed up. On the other hand, when the characteristics of the model prevent saturation to use up all primarily computed sub-state spaces, the algorithm exploits the additional resources to do this parallel.

In the future we want to extend our algorithm with heuristics to lead the parallelization, especially the order of directions the algorithm computes parallel. By leading the parallelization we expect additional speed up.

We would like to exploit the computational power of network of workstations and we will combine our parallel algorithm with distributed algorithms.

References

[1] T. Murata, Petri Nets: Properties, Analysis and Applications, *Proceeding of the IEEE*, 1989, Vol. 77, No.4, 541-580

[2] Miller D. M. and Drechsler R., Implementing a Multiple-Valued Decision Diagram Package., 1998, *The 28th International Symposium on Multiple-Valued Logic*.

[3] D. M. Miller and R. Drechsler, "On the construction of multiple-valued decision diagrams," *Proc. 32nd Int. Symp. on Multiple-Valued Logic*, pp. 245-253, May 2002.

[4] Ciardo, G., R. Marmorstein, and R. Siminiceanu., "The saturation algorithm for symbolic state-space exploration.", 2005, *International Journal on Software Tools for Technology Transfer* 8(1): 4-25.

[5] Ezekiel, Jonathan, G. Lüttgen, and R Siminiceanu., "Can saturation be parallelised? On the parallelisation of a symbolic state-space generator.", 2006, *PDMC conference on Formal methods: Applications and technology*: 331-346.

[6] McMillan, K.L. 1993. "Symbolic model checking."

[7] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

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

[9] Barnat, Jiří - Brim, Luboš - Rockai, Petr. DiVinE Multi-Core -- A Parallel LTL Model-Checker. In Automated Technology for Verification and Analysis. Berlin / Heidelberg : Springer, 2008. ISBN 978-3-540-88386-9, pp. 234-239. 2008, Seoul.

[10] B. Bingham, J. Bingham, F. de Paula, J. Erickson, M. Reitblatt, and G. Singh, "Industrial Strength Distributed Explicit State Model Checking": International Workshop on Parallel and Distributed Methods in Verification (PDMC), 2010.

[11] U. Stern and D. L. Dill: Parallelizing the murphi verifier, International Conference on Computer Aided Verification, pp. 256–278., 1997.

[12] Grumberg O, Heyman T, Schuster A. A work-efficient distributed algorithm for reachability analysis. Formal Methods in System Design. 2006;29(2):157-175.

[13] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In third International Conference on Formal methods in Computer-Aided Design (FMCAD'00), Austin, Texas, November 2000.

[14] Ciardo, G. "Saturation NOW." First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. 272-281.

[15] Chung, M.-Y., and G. Ciardo. 2009. "Speculative Image Computation for Distributed Symbolic Reachability Analysis." Journal of Logic and Computation: 1-19.

[16] G. Ciardo and A. S. Miner, "Storage alternative for large structured state spaces," in *Proc. of the 9th Int. Conf. of Modeling Techniques and Tools for Computer Performance Evaluation*, 1997.

[17] Ciardo, Gianfranco, G. Lüttgen, and Radu Siminiceanu. 2001. "Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation." In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, p. 328–342