FORWARD SATURATION BASED MODEL CHECKING

András VÖRÖS
Advisor: Tamás BARTHA

I. Introduction

Formal methods are gaining importance as safety critical, distributed and embedded systems are becoming widespread. By using formal verification we can find errors or we can prove the correctness in an early stage of the design. Model checking is an automatic verification method to check discrete, finite state models. In the last 20 years many model checking algorithms appeared: in this paper we focus on one of them, the so-called *saturation* algorithm. Saturation is a symbolic state space generation and model checking algorithm, which is efficient for globally asynchronous, locally synchronous (GALS) models. Former work presented structural model checking algorithms using saturation and constrained saturation, which were based on the classical backward traversal of the state space. In this paper we introduce saturation model checking based on forward state traversal. We hope this research direction to further improve the efficiency of model checking algorithms.

II. Preliminaries

Petri nets [1] are graphical models for concurrent and asynchronous systems, providing both structural and dynamical analysis.

An *event* in the system is the firing of an enabled transition. The firing of transitions is non-deterministic. The *state space* of a Petri net is the set of states reachable through transition firings.

In order to examine a model (for example a Petri net), we have to explore its possible dynamic behaviour, i.e. the state space. Traditional *symbolic state space exploration* uses encoding for the traversed state space, and stores this compact, encoded representation only. Decision diagrams proved to be an efficient form of symbolic storage, as applied reduction rules provide a compact representation form. Another important advantage is that symbolic methods enable us to manipulate large set of states efficiently.

A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f : \{0, 1, \ldots\}^K \to \{0, 1\}$. An MDD has a node set containing two types of nodes: non-terminal and two terminal nodes (0 and 1). The nodes are ordered into $K + 1$ levels. A non-terminal node is labelled by a variable index $0 < k \leq K$, indicating which level the node belongs to (which variable it represents), and has n_k (the domain size of the variable, in binary case $n_k = 2$) edges pointing to nodes in level $k - 1$ (the i-th edge of node n is written as $n[i]$). A terminal node is labelled by the variable index 0. Further information can be found in [2].

With the help of MDD based symbolic representation we are able to explore the state space of complex systems. 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 $(v_1, v_2 \ldots v_n)$, 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 $2n$ level decision diagram with variables: $\mathcal{N} = (v_1, v_2 \ldots v_n, v'_1, v'_2 \ldots v'_n)$, 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 using a breadth first search. The reachable set of states S from a given initial state s_0 is the *transitive closure* (in other words: the *fixed-point*)

of the next-state relation: $S = \mathcal{N}^*(s_o)$. Saturation based state space exploration 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 modelled with Petri nets.

The saturation algorithm consists of the following steps:

- *Decomposition:* Petri nets can be decomposed into local submodels. The global state is the composition of the components' local states: $s_g = (s_1, s_2, \ldots, s_n)$, where n is the number of components, and s_n is the local state of the n-th component. This decomposition is the first step of the saturation algorithm.
- *Event localization:* As the effects of the transitions 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* by the top (top_e) and bottom (bot_e) levels (submodels). Outside of this interval we omit the event e from the exploration.
- *Special iteration strategy:* Saturation iterates through the MDD nodes and generates the whole state space representation using a node-to-node transitive closure. In this way saturation avoids the peak size of the MDD to be much larger than the final size, which is a critical problem in traditional approaches. Let $\mathcal{B}(k, p)$ 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 $\mathcal{E}$ 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, iff node $\mathcal{B}(k, p) = \bigcup_{\forall e \in \mathcal{E}} \mathcal{N}_e^*(\mathcal{B}(k, p))$. The state space generation ends when the node at the top level becomes saturated, so it represents the state space: $S = \mathcal{N}^*(s_0)$.
- *Encoding of the next-state function:* Saturation algorithm uses a disjunctive-conjunctive transition relation decomposition [3], where the global next state relation $\mathcal{N}$ is constructed as the disjunction of the transition relations for all event: $\mathcal{N} = \bigcup_{\forall e} \mathcal{N}_e$. Each transition relation $\mathcal{N}_e$ is then constructed as the conjunction of sub-relations. Sub-relations are constructed model dependently, as the chosen higher level model determines how they can be efficiently created. In the case of ordinary Petri nets conjunctive representation can be based on *Kronecker matrices* [2], however for scalability and flexibility reasons we employ symbolic representation of the Next-state functions.
- *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 e events where $top_e = 1$. After finishing the first level, we saturate all nodes at the second level by firing all events, where $top_e = 2$. If new nodes are created at the first level by the firing, they are also saturated recursively. The procedure 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 or symbolically in MDD-s.
- *State space representation as an MDD:* A level of the MDD generated during saturation represents the local state space of a submodel. The possible states of the submodel constitute the domain of the variables in the MDD. Each local state space is encoded in a variable.

Model checking [4] is an automatic technique for verifying finite state systems. Given a model defined for example as a Petri net in our context, model checking decides whether the model fulfils the specification. Formally: let M be a Kripke structure (i.e. state–transition graph). Let f be a formula of temporal logic (i.e. the specification). The goal of model checking is to find all states s of M such that $M, s \models f$ ("$\models$" means "satisfies").

State space generation serves as a prerequisite for the structural model checking: verifying temporal properties needs the state space and transition relation representation. *CTL* (Computation Tree Logic) is widely used to express temporal specifications of systems, as it has expressive syntax and there are

efficient algorithms for its analysis. Operators occur in pairs in CTL: the *path quantifier*, either A (on all paths) or E (there exists a path), is followed by the *tense operator*, one of X (next), F (future, or finally), G (globally), and U (until). However we only need to implement 3 of the 8 possible pairings due to the duality [4]: EX, EU, EG, and we can express the remaining with the help of them in the following way: $\mathsf{EF}p \equiv \mathsf{E}[true\ \mathsf{U}p]$, $\mathsf{AX}p \equiv \neg\mathsf{EX}\neg p$, $\mathsf{AG}p \equiv \neg\mathsf{EF}\neg p$, $\mathsf{AF}p \equiv \neg\mathsf{EG}\neg p$, $\mathsf{A}[p\ \mathsf{U}\ q] \equiv \neg\mathsf{E}[\neg q\ \mathsf{U}\ (\neg p \wedge \neg q)] \wedge \neg\mathsf{EG}\neg q$.

The CTL model checking algorithm efficiently utilizes the data structures created previously, during the state space exploration. As CTL operators express next-state relations and fixed point properties, we have to efficiently express the inverse of the next-state function $\mathcal{N}^{-1}$. The semantics of the 3 CTL operators:

- **EX**: $i^0 \vDash \mathsf{EX}p$ iff $\exists i^1 \in \mathcal{N}(i^0)$ s.t. $i^1 \vDash p$. This means that EX corresponds to the inverse $\mathcal{N}$ function, applying one step backward through the next-state relation, formally: $\mathsf{EX}\, p = \mathcal{N}^{-1}(p)$
- **EG**: $i^0 \vDash \mathsf{EG}p$ iff $\forall n \geq 0, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash p$ so that there is a strongly connected component containing states satisfying p. This computation needs a greatest fixed-point computation: $\mathsf{EG}p$ = **gfp** $Z[p \wedge \mathsf{EX}\, Z]$
- **EU**: $i^0 \vDash \mathsf{E}[p\ \mathsf{U}\ q]$ iff $\exists n \geq 0, \exists i^1 \in \mathcal{N}(i^0), \ldots, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \vDash q$ and $i^m \vDash p$ for all $m < n$. Informally: we search for a state q reached through only states satisfying p. The computation of this property needs a least fixed-point computation: $\mathsf{E}[p\ \mathsf{U}\ q]$ = **lfp** $Z[q \vee (p \wedge \mathsf{EX}\, Z)]$

As it is easy to see, these operations and fixed-point calculations are based on the *pre-image* (inverse Next-state) computation operator: $\mathcal{N}^{-1}$. The question comes naturally: can we replace backward traversal based operators? The idea of forward state traversal symbolic model checking appeared to replace backward state traversal.

In order to be able to do forward model checking, we have to convert the semantics of the backward model checking [5]. If we examine a model with initial state s_0, where exactly predicate p_0 is true, so that $s_0 \vDash f$ can be rewritten: $s_0 \vDash f \iff p_0 \wedge f \neq \mathit{false} \iff p_0 \wedge \neg f = \mathit{false}$. The semantic of forward and backward traversal model checking differs, like the expressible possible properties [6]. Forward model checking is built on path expressions, which is contrary to the approach used by backward structural model checking. The forward model checking approach builds a so-called *path set expression (PSE)* and checks its validity in the model [7]. The basic elements of PSE-s are the following (p and q are propositional formulas):

- $[p]$ matches every one-step sequence, which satisfies p
- $[p]^*[q]$ matches every finite sequence, which ends in a state satisfying q, and all states before satisfies p. This is the forward traversal counterpart of the $\mathsf{E}[p\mathsf{U}q]$ CTL operator.
- $[p]^\omega$ matches every infinite sequence such that each state satisfies p. This is the forward traversal counterpart of the $\mathsf{EG}\ p$ CTL operator.
- $\alpha\beta$ matches to every sequence, such that the first finite part matches formula α, and after it the (tail) sequence matches β
- $\alpha : \beta$ matches to every sequence, such that the first finite part matches formula α, then there is a state, where both α and β is true, then the last (tail) sequence matches β

The main forward traversal evaluation procedures and their semantics are the following:

- $\mathsf{fw}([p][f])$: it computes the PSE $[p][f]$, the procedure computes $\mathcal{N}(p) \wedge f \neq false$,
- $\mathsf{fwU}(p, q)$: it computes the PSE $[p] : [q]^*$, the procedure computes the forward least-fixed point **lfp** $Z[p \vee \mathcal{N}(Z \wedge q)]$
- $\mathsf{fwG}(init, p)$: it computes the PSE $[init] : [p]^\omega$, the procedure computes the forward greatest fixed-pont **gfp** $Z[p \wedge \mathcal{N}(Z)]$; to be able to compute this greatest fixed-point, we have to compute the reachable states from $init$ state satisfying p: **lfp** $Z[init \vee \mathcal{N}(Z \wedge p)]$, which can be done with the formerly defined procedure: $\mathsf{fwU}(init, p)$

III. Saturation based forward model checking

Our aim is to apply saturation in forward model checking, so we need saturation based fw, fwU and fwG procedures. At the end of this section we show how CTL expressions are computable with these forward traversal procedures.

For this reason, we employ the so-called constrained saturation algorithm [8], with small modifications. The traditional constrained saturation algorithm uses $\mathcal{N}_l^{-1}$ to explore the states, so it uses backward state traversal. By changing the direction, and using $\mathcal{N}_l$ in the algorithm $ConsSaturate(p, q)$ [8], it will compute exactly the same states as procedure $\mathsf{fwU}(p, q)$. The main advantage of this approach is that we can avoid the intersection operations which are applied in the traditional approaches at every breadth-first step. So we have an algorithm to compute $\mathsf{fwU}(p, q)$. We have to be able to compute $\mathsf{fw}([p][f])$ and $\mathsf{fwG}(p)$ too to be able to handle more CTL operators. $\mathsf{fw}([p][f])$ needs a Next-state computation, which does not use saturation, it can be done with former approaches. The algorithm computing $\mathsf{fwG}(init, p)$ starts with computing $ConsSaturate(init, q)$, and then by standard breadth first search it computes the greatest fixed-point **gfp** $Z[p \wedge \mathcal{N}(Z)]$.

In the following the basic CTL expressions and their forward traversal based counterparts are:

- Forward EX evaluation: It can be changed to $\mathsf{fw}([p][f])$. So we can replace the outermost EX evaluation with Next-state computation, $\mathsf{fw}([p][f]) : p \wedge \mathsf{EX}f \neq \mathit{false} \iff \mathcal{N}(p) \wedge f \neq \mathit{false}$
- Forward EU evaluation: It can be changed to $\mathsf{fwU}(p, q)$, so we can replace the outermost EU evaluation as follows: $p \wedge \mathsf{E}[q \ \ \mathsf{U} \ \ f] \neq \mathit{false} \iff \mathsf{fwU}(p, q) \wedge f \neq \mathit{false}$
- Forward EG evaluation: Using the fwG operator, we can replace the outermost EG evaluation as follows: $p \wedge \mathsf{EG}\ q \neq \mathit{false} \iff \mathsf{fwG}(p, q) \neq \mathit{false}$

This way many of the CTL expressions are convertible to model check in a forward manner. However, there are still some examples, which are not, according to the literature [6]. The theoretical question is still open: which formulas are expressible with this logic [6].

IV. Conclusion

In this paper I have presented how forward traversal based CTL model checking can be done with the help of the saturation algorithm. The main motivation of this work is to further improve the efficiency of saturation by avoiding the full state space exploration which was necessary with former saturation based model checking algorithms.

In the future we plan to implement these algorithms and we would like to further improve EG computation, which seems to be the bottleneck of our approach.

References

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

[2] G. Ciardo and A. S. Miner, "Storage alternatives for large structured state spaces," in *Proceedings of the 9th ICCPE: Modelling Techniques and Tools*, pp. 44–57, London, UK, 1997. Springer-Verlag.

[3] G. Ciardo and A. Yu, "Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning," *Correct Hardware Design and Verification Methods*, 3725:146–161, 2005.

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

[5] H. Iwashita, T. Nakata, and F. Hirose, "CTL model checking based on forward state traversal," in *1996 IEEE/ACM International Conference on Computer-Aided Design*, pp. 82–87. IEEE, 1996.

[6] T. Henzinger, O. Kupferman, and S. Qadeer, "From pre-historic to post-modern symbolic model checking," in *Computer Aided Verification*, pp. 195–206. Springer, 1998.

[7] H. Iwashita and T. Nakata, "Forward model checking techniques oriented to buggy designs," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD) ICCAD-97*, pp. 400–404, 1997.

[8] Y. Zhao and G. Ciardo, "Symbolic CTL model checking of asynchronous systems using constrained saturation," in *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pp. 368–381, Berlin, Heidelberg, 2009. Springer-Verlag.