OPTIMIZING SATURATION BASED MODEL CHECKING

András VÖRÖS
Advisor: Tamás BARTHA

I. Introduction

Formal methods are widely used for the verification of safety critical and embedded systems. However, nowadays the use of formal methods is not limited to these areas as the demand for proven functional correctness increased in many other fields, e.g. in telecommunication and automotive industry. The main advantage of formal methods compared to extensive testing is that they can provide a mathematical proof for the correct behavior of the system, or they can prove that the system does not meet its specification. On the other hand, testing can only examine a portion of the possible behaviors. Usually the system undergoes testing after the implementation phase; formal methods can help finding bugs and design errors earlier, in the design phase. This makes the correction cheaper, and the development process more effective.

Nevertheless, formal verification is a complex task. As systems become more complex, the excessively growing number of possible behaviors of them leads to enormous memory or time consumption, so that the completion of the analysis process becomes impossible due to the limitations of the computational architecture. This is called the state space explosion problem.

One of the most prevalent techniques in the field of formal verification is model checking, which is an automatic technique to check whether a system fulfills the requirements. There are several model checking algorithms and approaches, starting from explicit traversal through the use of different reduction techniques and ending at the abstract methods. However, all methods need to generate a representation of the state space in order to run some analysis on it. Storing the state space representation can turn out to be quite difficult in cases where the state space explodes.

There are two main reasons behind this problem. The first reason is that the independently updated state variables lead to exponential growth in the number of the system states, even in the case of Boolean variables. The second reason is the asynchronous characteristic of distributed systems; the composite state space is often the Cartesian product of the local components' state spaces.

There are different ways to efficiently handle state space explosion. We can reduce the memory and time consumption with intelligent state traversal – we only traverse and store a representative set of the state space, which preserves the examined properties.

To overcome the difficulties caused by the large number of state variables, the use of symbolic methods is widespread. These algorithms can avoid storing state space explicitly; instead they encode the states and store them implicitly for example in decision diagrams. Symbolic methods enable us to efficiently store huge sets in memory. In general: symbolic techniques were a good choice at the register transfer level (RTL) design, where the large number of variables lead to large memory consumption. At system level, asynchronous components lead to large state spaces, dealt with reduction. This also shows the different strengths of the algorithms. In this paper I introduce a special symbolic model checking algorithm, called saturation, and I examine the effect of some heuristics to its performance.

II. Symbolic state representation and saturation

Traditional symbolic model checking use encoding to store the traversed state space, and stores this compact representation only. Decision diagrams proved to be a proper solution for this purpose, as applied reduction rules provide a compact representation form. In the early times Binary Decision Diagrams were the most common, but in the last decade many other variants appeared to comply

with the broadening application requirements. From this wide area I introduce Multiple Valued Decision Diagrams (MDD), which I present in the remaining of the paper.

Multiple Valued Decision Diagram is a directed acyclic graph with a node set containing 2 types of nodes ordered into levels: non-terminal and 2 terminal vertices. A non-terminal vertex is labeled by a variable index *k*, which indicates to which level belongs the node (which variable is represented by it), 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 in level *k* two nodes have identical successors, they are also identical. Redundant nodes are allowed, so 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.

The first step in symbolic model checking is to encode the possible states. Traditional approach encodes each state with a given variable assignment (v_1, v_2... v_n) and stores it in a decision diagram. To encode the possible state changes, we have to encode the transition relation, which can be done in a *2n* level decision diagram with variables: (v_1, v_2 ...v_n, v'_1, v'_2...v'_n), where the first *n* variables represent the "*from"*, and variables (v'_1, v'_2...v'_n) the "*to"* states. The state space traversal builds this during a breadth first search, or in a combined breadth first and depth first traversal called chaining.

The main motivation of saturation state space exploration [1] is to combine symbolic methods with a special iteration strategy, which proved to be very efficient for asynchronous models. These models consist of some components interacting with each other less often than executing their local transitions. In this case, we can decompose the model to local sub-models, and 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. This decomposition is the first step of the saturation algorithm. Saturation needs the so called *Kronecker* consistent decomposition, 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: $\mathcal{N}_e = \mathcal{N}_{1,e} \times \mathcal{N}_{2,e} \times \ldots \times \mathcal{N}_{n,e}$. Note that when modeling 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. Many modeling languages, such as Petri nets have this property [1].

During saturation, decomposition serves the basic of the symbolic encoding – we code in a variable a sub-model's local states. We need as many variables as many sub-models we divided the model into. Opposite to the traditional approach, the *Next-state* function is not coded explicitly in a *from-to* relation in the MDD, instead we store to each sub-model the local *Next-state* function in the so called *Kronecker* matrix [2]. As the transitions usually have only local effects, 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 don't deal with it. *Kronecker* matrices and local state spaces are built dynamically during the traversal, while affected level intervals of events (top_e, bot_e) are defined before, from structure.

Saturation is a special iteration strategy, which consists of a bottom-up building of the decision diagram with a special depth first traversal of the state space. The main aim is to explore the sub-models local state space in a greedy manner, and encode them before other sub-models are explored. This is similar to the chaining method; the main difference is that saturation localizes the effects of the changes in the MDD, updating nodes instead of updating the whole MDD.

The reachable set of states $\mathcal{S}$ from a given initial state s is the closure of the *Next-state* relation: $\mathcal{S} = \mathcal{N}^*(s)$. Saturation iterates through the MDD nodes and reaches the whole state space representation from node to node transitive closure generation. 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. 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 ε be the set of events affecting *k*. level and below, so where $top_e \leq k$. We call a node *p* at level *k* saturated, if node $\mathcal{B}(k,p) = \cup_{\varepsilon} \mathcal{N}_e^*(\mathcal{B}(k,p))$. The state space generation ends when the node at the top level becomes saturated,

so it represents $\mathcal{S} = \mathcal{N}^*(s)$ for all events.

As saturated nodes represent the local fixed-points for events having effects localized to the sub-graph, only saturated nodes can appear in the final state space representation [2]. In addition, updating nodes during the traversal can be made locally, called in-place update, just setting the edges to the newly explored sub-MDD. So we need fewer operations affecting the whole MDD than traditional approaches, instead we make operations with smaller impact. This property helps keeping the peak size of the MDD low during the state space exploration.

State space generation serves as a prerequisite for the structural model checking: verifying temporal properties needs the state space and transition relation representation. CTL is widely used to express temporal specifications of systems, as it has expressive syntax and there are efficient algorithms for its analysis. In CTL, operators occurs in pairs: 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, from the 8 possible pairings due to the duality [1], we only need to implement 3: *EX*, *EU*, *EG*, and we can express the remaining with the help of them. These operators also benefit from the locality exploited by saturation. The semantics of them are:

- ***EX***: $i^0 \models EX\ p$ iff $\exists\ i^1 \in \mathcal{N}(i^0)$ s.t. $i^1 \models p$ ("|=" means "satisfies"). This means that *EX* corresponds to the inverse $\mathcal{N}$ function, applying one step backward through the *Next-state* relation, using transposed *Kronecker* matrix. This way we can exploit locality efficiently.
- ***EG***: $i^0 \models EG\ p$ iff $\forall n \geq 0, \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \models p$ so that there is a strongly connected component containing states satisfying p. This computation needs a greatest fixed-point computation, so that saturation cannot be applied for it. Computing the closure of this relation however profits from the locality accompanying the decomposition.
- ***EU***: $i^0 \models E[pU\ q]$ iff $\exists n \geq 0, \exists i^1 \in \mathcal{N}(i^0), ..., \exists i^n \in \mathcal{N}(i^{n-1})$ s.t. $i^n \models q$ and $i^m \models p$ 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, which can exploit the efficiency of saturation.

However, before performing saturation in *EU*, we have to classify events into categories in order to define the breadth first and the saturation based steps in the fixed-point calculation:

- An event *e* is **dead** with respect to a set of states *S* if $\mathcal{N}_e^{-1}(S) \cap S = \emptyset$, these events are omitted from the fixed-point calculation.
- An event *e* is **safe**, if it cannot lead from outside *S* to sates in *S*, formally: $\emptyset \subset \mathcal{N}_e^{-1}(S) \subseteq S$.
- All other events are **unsafe**.

With the help of this categorization, we decompose the fixed-point calculation into 2 steps:

- Computing the closure of relations of the safe events can be efficiently done by saturation
- By breadth-first traversal the algorithm explores unsafe events. As from states reached by unsafe steps we have to filter out those, which do not satisfy *p* or *q*, we have to compute the intersection of them with $p \cup q$. This intersection is evaluated in all iteration.

The efficiency of *EU* computation highly depends on the efficiency of the saturation steps, because the number of breadth first steps (and intersection operations) depends on the model and the temporal logic formula itself, so we can only reduce the runtime of the saturation.

Time and memory consumption of saturation depends on the iteration order, and of course on the size of the MDD. Measurements showed that the size of the *Next-state Kronecker* matrices is just a small fraction of the whole memory required by the algorithms.

Iteration order and MDD size are not independent. Actually, the algorithm iterates through all nodes, so the complexity is at least the number of nodes in the MDD. We have to point out that there are more iterations, usually by a polynomial factor. As the size of the MDD can be exponential in the size of the variables, this is a critical point in symbolic model checking. Good variable ordering is the key in MDD size reduction. Finding good variable order is an NP hard problem, however there are usable heuristics that perform well.

III. Optimization

The main motivation of the optimization is to enable the algorithm to exploit locality efficiently, which is important not only because of the number of the iteration steps, but the size – and of course, the efficiency – of the MDD operations. In addition, the cached values of the event-node pairs – which enable the algorithm to avoid redundant operations on MDD nodes – couldn't be used efficiently, when the iteration order, so that the MDD variable order doesn't reflect the structure of the system. Optimizing saturation consists of the decomposition of the model into sub-models, and then the algorithm assigns an MDD variable to each of them. Good decomposition, i.e. when functionally dependent sub-models are composed into the same variable can reduce the size of the MDD [2]. However, coding big state spaces into a single variable leads soon to an unusable algorithm as it converges to the explicit state space representation method.

There are two main trends in order to improve the efficiency of the saturation iteration strategy [2]: choosing the ordering of the variables to make the effects of the transitions locally or to make the top levels of the transitions effects lower. First approach improves locality by trying to reduce the distance between the top and bottom levels of the transitions, so that we minimize the sum of transition spans. Second approach uses the fact that saturation builds the MDD representing $\mathcal{S}$ in a bottom-up fashion, so placing transitions as low as possible is the preferable. This approach minimizes the sum of the transition tops. Combining the approaches is a good compromise.

The literature about MDD size reduction is wide. These approaches use some kind of variable reordering method. The basic operation is the so called "adjacent level interchange", which changes the order of the adjacent variables, trying to reduce their sizes. This way we can reduce the size of the MDD step-by-step, and avoid overrunning the memory limit. The optimization algorithms perform these steps until a good variable order is found. However, in this context, this method can only be used after the state space generation, so it cannot make saturation faster.

Reducing the size of the MDD needs some heuristics to find a good variable order before the generation. If we use some knowledge about the model it can help us to avoid mistakes which enlarge the MDD. It is well known, that if there are concurrent sub-models, the variables representing their states shouldn't be overlapped. Consider 2 components with variables x_1, x_2 and y_1, y_2, where functional dependencies ($\rightarrow$) occur inside the components. In this case there is no functional dependence: $(x_1, x_2) \rightarrow (y_1, y_2)$, only $(x_1) \rightarrow (x_2, y_1, y_2)$ and $(x_1, x_2, y_1) \rightarrow (y_2)$. Using the variable order x_1, y_1, x_2, y_2, 2 dependencies remain: $(x_1) \rightarrow (y_1, x_2, y_2)$ and $(x_1, y_1, x_2) \rightarrow (y_2)$, however we create a new dependence: $(x_1, y_1) \rightarrow (x_2, y_2)$, which will increase the number of nodes. Let me point out, that these functional dependencies mean some transitions in the model checking context, which control the possible reachable states, so the heuristics used for the iteration order indeed localizes the dependencies in the MDD making it smaller.

Using heuristics from [2] in the structural CTL model checking may improve the performance. As *EU* and *EF* computation builds mostly on saturation, they can exploit heuristics after the state space generation. However the algorithm was optimized also before state space generation, applying the heuristics during the model checking, there were a 3-5% additional performance gain. Applying MDD size reduction before *EG* computation decreased the computation time with 5-50%.

IV. Conclusion, further work

The usage of static variable ordering in CTL model checking proved its usefulness. In the future I would like to extend the algorithms to apply dynamic reordering during saturation.

References

[1] Ciardo, G., R. Siminiceanu.: "Structural symbolic CTL model checking of asynchronous systems." *CAV,* 2003

[2] Ciardo, G., G. Lüttgen, A.J. Yu.: "Improving static variable orders via invariants." *ICATPN 2007,* Springer