MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Trading space for time in explicit-state model checking

DIPLOMA THESIS

Pavel Mičan

Brno, Spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Pavel Mičan

Advisor: Jiří Barnat

Acknowledgement

If the bricks aren't well made, the wall falls down.

This is an awfully big wall I'm building here, so I need a lot of bricks. Fortunately I know a lot of brickmakers, and all sorts of other folks as well.

[George R.R. Martin]

Thanks and appreciation to all good people who guided me through the tough time of understanding the depths of DiVinE model checker, to those who stood by me in my frequent states of despair and my special thanks to my supervisor Jiří Barnat, who must have grown hair only to lose them again waiting for my progress.

Abstract

To reduce the memory size requirements in model checking we came with an approach of storing only certain states based on their ordering. A random assignment of an integer value (ideally unique) is given to each state and while processing en edge of a graph only those states are stored which have its assigned value lower to the maximum value found since the last state had been stored. This simple solution trades memory space for computing time and is applicable to various DFS and even BFS on-the-fly algorithms. In this thesis I present the background of the model checking and the state explosion problem together with detailed description of the technique.

Keywords

model checking, verification, distributed, to store or not to store, OWCTY, Nested DFS, DiVinE, state space explosion

Contents

1 **Introduction** 2
2 **Model Checking and State Explosion Problem** 4
2.1 *System model* 4
2.2 *LTL formulæ* 5
2.2.1 LTL definition 5
2.2.2 LTL into Büchi 6
2.3 *The Big Bang* 6
3 **Algorithm** 8
3.0.1 altering OWCTY algorithm 8
4 **Conclusions** 12

1 Introduction

Model checking [12] is a well established method of formal verification with increasing popularity. The use of Model checking is becoming crucial for verifying correctness properties of hardware and software systems or protocols that require high level of reliability, and it has become an integrated part of the verification process in many companies such as IBM who maintain RuleBase symbolic model checking tool [9, 14], Microsoft's Static Driver Verifier (SDV) [20] developed as a part of the SLAM project [5] that recently successfully demonstrated a computer-aided verification on real programs [4, 6], well known SPIN tool developed at Bell Labs [19], and many others[1, 2, 13, 24], and in addition to all these a DiVinE model checker[8, 15] is being developed in the laboratory of Parallel and Distributed Systems (ParaDiSe) [18].

At the moment the DiVinE model checking tool provides multiple different algorithms that traverse given graph and are capable of finding an accepting cycle in order to prove correctness of a given property in a modelled system or provide not only the information on it's misbehaviour, but also a counterexample in a form of a succession of steps that lead from the initial state to a state where the conditions are not met. Apart from the classical NestedDFS algorithm, the DiVinE exploits breadth-first-search (BFS) algorithms such as Maximal Accepting Predecessor (MAP) or One-Way Catch Them Young (OWCTY)[7]. Detailed description to these algorithms is discussed in Chapter 3.

Having said that, the industrial-scale models are mainly restricted by memory size. For the past years the research in model checking was mainly concerned with a problem commonly known as a state-space expansion. The issue is to deal with a memory requirements which grow exponentially with the size of tested system and/or property[16]. The chapter 2 is dedicated to explain what the explicit state space model checking is and how does a property checking lead to the expansion problem.

The aim of the technique presented in this work is to provide a simple solution to reduce memory consumption that would be independent of an algorithm used to traverse the given graph and also allow the DiVinE model checker use it's distributing algorithm potential. To do that, the method used must be able to work with on-the-fly algorithms, ie. algorithms that dynamically pass through a given graph searching for an accepting cycle. To represent it's effectiveness, I tested the code implemented in DiVinE using two DFS algorithms, namely NestedDFS and reachability,

and MAP together with OWCTY were used as BFS algorithm representatives. Yet, the results are not a part of this edition for the lack of their credibility, for the implementation at the moment does not ensure the correct behaviour and in various models triggers segmentation faults.

At the beginning of the work I present a brief insight into the Model Checking phenomenon and explain the procedure leading to the state explosion problem.

The third chapter is then fully dedicated to the new technique.

At the very end of the thesis assumptions on further expansions of the technique and possible research are stated.

2 Model Checking and State Explosion Problem

To address the problem of state explosion in LTL model checking it is crucial to understand two essential structures. First, the model of a tested system represented as Büchi automatons (BA)[21], and second the Linear Temporal Logic (LTL) expressions again transformed into a BA.

Having these two basic structures, we may claim that a model satisfies LTL formulæ if and only if the language of the system model is a subset of the language of a given LTL formula. This conclusion is described in more detail at the end of this chapter in the Big Bang section. But first, let me introduce the two basic structures.

2.1 System model

The system model is an oriented graph representation of the system we'd like to test. The model is based on Kripke Structure (KS) [3, 12], which is a variation of a non-deterministic automaton.

The model being a representation of a program, it needs to reflect basic behaviour and properties of each state in which the running program can be. Thus, the Kripke structure is a four-tuple (S, T, I, S_0) where:

- S is a finite set of states in which the system can be,
- $T \subseteq S \times S$ is a transition relation that must be total, and represents transitions from one state to another,
- $I : S \to 2^{AP}$ is an interpretation that assigns each state a set of valid atomic propositions,
- S_0 is a set of initial states.

The interpretation I in the structure expresses certain qualities of inner state properties. An example of an atomic proposition might be strictly defined, eg. "$x = 4$" implies that in the state containing this AP the value of x is exactly four, but AP might even assert a range of possible values, eg. "$x < 5$" or "$min(x, y, z) > 0$". In general, an atomic proposition is any statement that can be easily tested by a machine and restricts values of properties that appear in the modelled system.

The Kripke Structure can be easily transformed into a Kripke Transition System by adding fifth element $L : T \to Act$ of labels. The labels assign a set of actions Act the program may execute to each transition.

Having the system structure defined there is only a small step needed to be able to automatically decide whether the system run has desired quality.

The formalism of Büchi Automaton (BA) gives us an ability to walk through the infinite system and decide the acceptance. Let (S, T, I, S_0) be a Kripke structure, then the BA $(\Sigma, S_B, s_0, \delta, F)$ is constructed as follows:

1. $\Sigma = 2^{AP}$
2. $S_B = S \cup \{s_0\}$
3. s_0 is a new initial state, ie. $s_0 \cap S = \emptyset$
4. δ is defined as follows:
   - $(s, \alpha, s') \in \delta fors, s' \in Siff(s, s') \in T \wedge \alpha = I(s')$ and
   - $(s_0, \alpha, s) \in \delta iffs \in S \wedge \alpha = I(s)$
5. $F = S \cup \{s_0\}$

2.2 LTL formulæ

In this section I will describe in short what the LTL formula is and then outline the process of its' transformation into a Büchi automaton.

2.2.1 LTL definition

Using the Linear (or Linear-Time) Temporal Logic expressions one can describe assertions of system paths. In Temporal Logic we define following operators:

- $F\varphi$ – somewhere in the **F**uture the φ is satisfied
- $G\varphi$ – φ is **G**lobally satisfied
- $\varphi U\psi$ – ψ is certainly satisfied at some point but **U**ntil then the φ is satisfied
- $X\varphi$ – φ is satisfied in the ne**X**t state
- $\varphi W\psi$ – **W**eakens the Until operator by the fact that ψ may not be satisfied at any point
- $\varphi R\psi$ – ψ is satisfied until it's **R**eleased by concurrent satisfaction of $\psi \wedge \varphi$

An obvious question arise at this point. What are these φ and ψ formulæ? From what I already described, the answer is intuitive. We already know the set of Atomical Propositions. Now we can easily claim that every $p \in AP$ is an LTL formula and it's satisfaction is trivially decided against each state in the system. Then we may iteratively define *negation* and logical *or* as $\neg\varphi$ and $\varphi \vee \varphi$ being formula. The semantics of these is intuitive.

For simplicity of use, the LTL generally used is restricted to ne*X*t and *U*ntil operators. It can be proven that all the others can be described using

these two [3]. Thus the summarized LTL syntax is:

$$\varphi ::= p|\neg\varphi|\varphi \vee \varphi|X\varphi|\varphi U\varphi$$

2.2.2 LTL into Büchi

To create a Büchi representation of the LTL formula, all we need to achieve is to construct a simple automaton model, and this model is nothing else than a run of Kripke structure. That may be intuitively described as a succession of states with such APs which reflect all possible behaviour satisfying the formula.

The exact process of transforming the formula exceeds the dimension of this thesis, but in short, the process of the translation follows these simple steps:

1. conversion of an LTL formula into a normal form
2. computation of a transition graph
3. conversion of the transition graph into a generalised BA (GBA)
4. conversion of GBA into a BA

It can be shown that all the valid LTL formulæ may be transformed into the BA following these steps[3, 23].

2.3 The Big Bang

In the beginning of this chapter I mentioned that the system model satisfies the given LTL formula *iff* the language of the former is a subset of the second. Let me introduce this phenomenon in more detail and finally hit the fan with an immense explosion!

We already have the formal description of LTL and System BA. From the automaton theory[17] we know that an automaton accepts words of a certain language, thus fully defines the language. The language accepted by an automaton A is denoted as $L(A)$. The crux of all the model checking is to decide whether the system model M satisfies given formula φ. As to follow the convention, let us call the formula a *property* from now on. From what we already know it's only natural to say that the following must be true:

Let M be the modelled system, φ the given property to be verified, L_{sys} the language accepted by the system BA ($L(A_{sys})$), and L_φ the language accepted by the property BA ($L(A_{LTL})$) then

$$M \models \varphi \iff L_{sys} \subseteq L_\varphi$$

and after a few elemental logical transformations we may easily convert the problem onto a test of a language emptiness:

$$M \models \varphi \Longleftrightarrow L(A_{sys}) \times L(A_{\neg\varphi}) = \emptyset$$

The result of the combination of these two automatons is called a *product* and it's size entails the State Explosion Problem. The fact is that even with relatively small system and property automatons, the product may have thousands of states[3, 7, 11, 12, 16, 22], and when we expect to verify most of the real-life systems, the memory consumption really matters and becomes the major bottleneck to the point of inability to use any model checker.

3 Algorithm

In this chapter I'll present the dynamic algorithm, that should slightly reduce the memory consumption while traversing the product graph.

To find a cycle in a given graph, we only need to store a single vertex on the accepting cycle as proven in [10]. It may be intuitive for a trained eye, that while searching for an accepting cycle, the single condition necessary to decide it's existence is to find at least one previously visited state from which the currently tested accepting state is reachable.

Let $G(V, E, L_{abels})$ be a graph where V is a set of vertices, $E : V \to V$ edges and $L_{abels} : V \to Int$ a random assignment of a single integer value to each vertex $v \in V$.

Statement 3.0.1 *Let $\geq$ be a total order relation over V, then while resolving a transition $s \to s'$ while searching for an accepting cycle in the graph G using on-the-fly algorithm, we may store only such states s' that meet condition $s \geq s'$ without any impact on algorithm correctness.*

The correctness of this statement seems to be only partially correct. The Depth-First Search (DFS) algorithms such as the well known Nested-DFS algorithm, seem to behave as expected without any further changes. Altering the storing condition to the BFS algorithms seem to suffer from a slightly disrupted behaviour that has to be corrected. In the following section the OWCTY algorithm is analysed in detail and the necessary changes are pinpointed and explained.

At this point a very valid question might pop up. Is the technique presented really useful, when algorithms themselves need to be altered to exploit it's contribution? This is surely a legitimate concern and my answer to this question would be yes, I still consider the technique to be universal, but at this point, I would like to postpone an attempt to explain the reason after at least one algorithm is examined in detail which would provide more essential background to decide.

3.0.1 altering OWCTY algorithm

The Algorithms 1,2,3 and 4 demonstrates the behaviour of modified OWCTY algorithm. All the lines that differ from the original algorithm are marked with an arrow.

Algorithm 1 modified OWCTY

```
1: procedure OWCTY
2:     INITIALIZE(
3:     repeat)
4:         oldSize ← |S|
5:         for all s ∈ V do
6:             s.pre ← ∅
7:         end for
8:         enqueue all states from S ∩ F into q
9:         REACHABILITY
10:        enqueue all states from S ∩ F into q
11:        ELIMINATION
12:    until |S| = oldSize return |S| > 0
13: end procedure
```

Algorithm 2 modified OWCTY

```
1: procedure INITIALIZE
2:     enqueue init into q
3:     while ¬q.empty do
4:         t ← q.pop()
5:         if t ∉ S then
6:             if hashCondition(t) || is Final(t) then          ▷ ←−
7:                 add t to S                                   ▷ ←−
8:             end if
9:             for all (t, u) ∈ E do
10:                enqueue u into q
11:            end for
12:        end if
13:    end while
14: end procedure
```

Algorithm 3 modified OWCTY

```
1:  procedure REACHABILITY
2:      S ← ∅
3:      while ¬q.empty do
4:          t ← q.pop()
5:          if t ∉ S then
6:              if hashCondition(t) || isFinal(t) then                ▷ ←—
7:                  add t to S                                        ▷ ←—
8:              end if
9:              for all (t, u) ∈ E do
10:                 if hashCondition(u) || isFinal(u) then            ▷ ←—
11:                     u.pre ← u.pre + 1                             ▷ ←—
12:                 end if
13:                 enqueue u into q
14:             end for
15:         end if
16:     end while
17: end procedure
```

Algorithm 4 modified OWCTY

```
1:  procedure ELIMINATION
2:      while ¬q.empty do
3:          t ← q.pop()
4:          if t.pre ≤ 0 then                                         ▷ ←—
5:              remove t from S
6:              for all (t, u) ∈ E do
7:                  u.pre ← u.pre − 1
8:                  enqueue u into q
9:              end for
10:         end if
11:     end while
12: end procedure
```

The main algorithm 1 does not suffer any changes at all. The lines 6-7 of algorithms 2 and 3 contain the crucial part that corresponds to the test for a *hashCondition(u)*, which represents the comparison of the ordering defined on all the states of the given graph. It is worth mentioning that the set S corresponds to what would be a hash table of states in any model checker implementing OWCTY algorithm. Thus postponing the insertion of the certain state into the set S reduces the amount of states that need to be stored.

On the line 11 of the REACHABILITY(p)rocedure the algorithm is slightly modified to reflect certain changes in behaviour. The initial state t of the transition pair (t, u) increases the number of predecessors only to those states that are stored and always by the value of 1. If there are more than one predecessors of t and the t is not stored, all the paths following the t vertex are walked through again until at least one vertex representative of each cycle is stored and accumulates all the single vertex increments, thus holding the correct number of predecessors when all the vertices from the queue q are processed.

Another change to the original algorithm needed to be made in the condition at the line 4 of the ELIMINATION(p)rocedure. The reason is that the t vertex in the Elimination phase is at that point stored only in the queue of unprocessed vertices, hence it might not be stored in the hash table and in that case we now allow to decrease an undefined predecessor counter to -1. Again as in the case of the REACHABILITY(p)rocedure the negative vertex that is not stored in the hash table may be inserted into the queue repeatedly with the negative value assigned, but the succession of such decrements will, by definition of the hashCondition, in finite number of steps affect one vertex that is stored in the table and correctly adjust the number of predecessors.

At this point I would like to return to the question that surfaced before I presented the algorithm. As we seen, the OWCTY algorithm had to be modified not only by the storing condition, but also at two points the original definition had to change the conditions to work correctly. But after closer examination, one can see that the change to the lines 10-12 of REACHABILITY(p)rocedure is only an optimization and the change to the line 4 in the Algorithm 4 would be functional even without the impact of this approach. The allowance of negative number is a simple generalisation of the former condition that couldn't occure and thus the most unambiguous definition was used.

4 Conclusions

The algorithm presented should provide certain memory usage reduction in trade with the computational time. Sadly the implementation is not yet at such a phase to be readily tested on real models. Few simple tests suggested that while using one simple hash function to encode state ordering, the memory savings varied in between 5 to 10% ran in the DivInE model checker. Nevertheless, the algorithm implemented still triggers various segmentation faults at the moment the correctness of its implementation and results obtained can not be trusted yet.

The future work hence imply an attentive insight into the application structure and correct inclusion of the algorithm, and of course test the product on various models to compare the measurements with the original ie. nonmodified behaviour.

Further work may be aimed at choosing various functions to generate generally better states ordering, and even combining multiple ordering mechanisms at once to ensure as much state storing skips as possible.

Bibliography

[1] Y. Abarbanel-Vinov, N. Aizenbud-Reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal, and T. Yatzkar-Haham. "On the effective deployment of functional formal verification". In: *Formal Methods in System Design* 19 (2001).

[2] A.Goel and W. Lee. "Formal verification of an IBM Coreconnect Processor Local Bus arbiter core". In: *37th Design Automation Conference (DAC)*. Association for Computing Machinery, Inc., June 2000, pp. 196–200.

[3] Christel Baier. *Principles of model checking*. Cambridge, Mass: MIT Press, 2008. ISBN: 9780262026499.

[4] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. "SLAM2: Static Driver Verification with Under 4% False Alarms". In: *Formal Methods in Computer Aided Design*. 2010. URL: `http://research.microsoft.com/en-us/projects/slam/slam2_fmcad2010_final.pdf`.

[5] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft*. Technical Report MSR-TR-2004-08. Microsoft Corporation, Jan. 2004.

[6] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM". In: *Commun. ACM* 54.7 (July 2011), pp. 68–76. ISSN: 0001-0782.

[7] Jiří Barnat, Luboš Brim, and Ivana Černá. "Cluster-Based LTL Model Checking of Large Systems". In: *Formal Methods for Components and Objects*. Ed. by FrankS. Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul Roever. Vol. 4111. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 259–279. ISBN: 978-3-540-36749-9.

[8] Jiří Barnat, Luboš Brim, Ivana Černá, and Pavel Šimeček. "DiVinE - The Distributed Verification Environment". eng. In: *In proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC05)*. Lisboa, Portugal: TU Munchen, 2005, pp. 89–94.

[9] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. "Model checking the IBM Gigahertz Processor: An abstraction algorithm for high-performance netlists". In: *11th International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 1999, pp. 72–83.

[10] Gerd Behrmann, Kim G. Larsen, and Radek Pelánek. "To Store or Not To Store". In: *Computer Aided Verification (CAV 2003)* (2003).

[11] Gerd Behrmann, Kim Guldstrand Larsen, and Radek Pelánek. "To Store or Not to Store". In: *CAV*. 2003, pp. 433–445.

[12] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT press, 1999. ISBN: ISBN 0-262-03270-8.

[13] C. Eisner, R. Hoover, W. Nation, K. Nelson, I. Shitsevalov, and K. Valk. "A methodology for formal design of hardware control with application to cache coherence protocols". In: *37th Design Automation Conference (DAC)*. Association for Computing Machinery, Inc., June 2000, pp. 724–729.

[14] Cindy Eisner. "Model Checking the Garbage Collection Mechanism of SMV". In: *CAV'01*. 2001.

[15] Masaryk University Faculty of Informatics. *DIVINE*. [Online; accessed 19-May-2013]. 2013. URL: `http://divine.fi.muni.cz/`.

[16] Moritz Hammer and Michael Weber. "To Store or Not To Store" reloaded: Reclaiming memory on demand". In: *Formal Methods: Application and Technology (FMICS'2006). LNCS*. Springer, 2006, pp. 51–66.

[17] John Hopcroft. *Introduction to automata theory, languages, and computation*. Boston: Addison-Wesley, 2001. ISBN: 0201441241.

[18] Faculty of Informatics, Masaryk University. *ParaDiSe*. [Online; accessed 19-May-2013]. 2013. URL: `http://paradise.fi.muni.cz/`.

[19] Bell Labs. *Spin – Formal Verification*. [Online; accessed 19-May-2013]. May 2013. URL: `http://spinroot.com/spin/whatispin.html`.

[20] Microsoft. *About Static Driver Verifier*. [Online; accessed 19-May-2013]. May 2013. URL: `http://msdn.microsoft.com/en-us/library/windows/hardware/gg487498.aspx`.

[21] Wolfgang Thomas. "Handbook of theoretical computer science (vol. B)". In: ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990. Chap. Automata on infinite objects, pp. 133–191. ISBN: 0-444-88074-7.

[22] Antti Valmari. "The state explosion problem". In: *Lectures on Petri Nets I: Basic Models*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Vol. 1491. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 429–528. ISBN: 978-3-540-65306-6.

[23] Wikipedia. *Linear temporal logic to Büchi automaton — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-May-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Linear_temporal_logic_to_B%C3%83%C2%BCchi_automaton&oldid=533165879.

[24] Wikipedia. *List of model checking tools — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-May-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=List_of_model_checking_tools&oldid=554949058.