Marking Pocket States for Bounded On-the-fly Model Checking

Wietze Spijkerman
w.spijkerman [at] student.utwente.nl

ABSTRACT

Model checking of state transition systems has been proven useful for proving absence of properties. However, model checking is a costly process due to heavy use of resources. On-the-fly model checking attempts to find counter examples faster. Bounded model checking offers a state space generation strategy for models of which termination is guaranteed. Combining these two approaches leads to redundant state space exploration during the emptiness check. This redundancy can be reduced by marking pocket states. Practical results however show that this reduced redundancy does not necessarily yield an increase in efficiency.

Keywords
automata-theoretic approach, bounded model checking, on-the-fly model checking, pocket state

1. INTRODUCTION

In the past twenty years model checking\(^1\) has gained interest and has become a proven approach towards automatic system verification. Model checking is a procedure that verifies whether a model of a system satisfies a certain property, expressed in a logical formula. There are different approaches towards model checking. These can be categorized in three groups: semantic approach, automata-theoretic approach and tableau approach\(^2\). This paper focuses on the automata-theoretic approach. This approach is useful when performing automated verification of a model.

Proving that a counter example is absent is only possible when the model describes a finite state space. The use of bounded model checking has the advantage of limiting the state space generated by the model and therefore guaranteeing termination of the exploration of the state space. This is done by defining a bound to which the state space is generated\(^3\) and offers an approach for the systematic analysis of infinite state spaces.

Recently developed model checkers, such as model checkers based on graph transformation, offer a more intuitive interface to create models and find logical errors\(^4\). In order to find these logical errors, on-the-fly model checking is applied to models. On-the-fly model checking combines both exploration and verification of a property while generating the state space. The advantage of this approach is that exploration and verification terminates as soon as a logical error is found, thus limiting resources and time used.

By combining on-the-fly model checking with bounded model checking, the mentioned advantages of both approaches are combined. However, this raises the problem of exploring and verifying previously explored state spaces. This is regarded as redundant work. This paper presents research on the subject of reducing this redundancy by using so-called pocket states. Pocket states are a characterization of previously explored states of which can be guaranteed that they do not lead to a counter example. Because of this characterization, paths leading to marked pocket-states do not have to be considered.

The paper is structured as follows: In section 2 the general approach of automata-theoretic approach based on emptiness checks is explained. Following this approach, bounded model checking and on-the-fly model checking and the combination of the two are placed in the context of this approach. The problem of redundancy that arises when combining bounded model checking with on-the-fly model checking is stated in section 3. Section 4 describes the requirements of pocket states and includes the detection and use of pocket states into an on-the-fly model checking algorithm. Section 5 states the components that are needed and used to carry out experiments, together with which experiments are carried out. In section 6 the experimental results are presented that follow to the conclusion and discussion in section 7.

2. BACKGROUND

Model checking consists of a formal description of a model of a system and a formal description of a property for which that model needs to be checked. The model defines the behaviour of the system. The property defines the desired behaviour for which the model is checked, expressed in a modal logic.

In order to perform model checking both the model and the verification property are combined and an emptiness check is done if a violation of the property occurs. An emptiness check takes the negation of the given property and checks whether an accepting run through the product automaton is present. If the combination of this property and the system yield a non-empty result it means that a counter-example that violates the property is found.

This general automata-theoretic model checking approach can be stated as the following five steps:

1. There is a model $M$ and a verification property $\phi$
2. An automaton $A_{\neg\phi}$ is obtained from the negated property $\neg\phi$
3. $M$ defines the state space $S$ of the system which is generated
4. The product automaton $P$ of $S \times A_{\neg\phi}$ is produced
5. If this product automaton P contains accepting runs $L(P) \neq \emptyset$, this language describes a counter example; otherwise the absence of a counter example is proven ($M \models \phi$).

This approach is depicted in figure 1. In figure 1 square boxes are data representations whereas the rounded boxes are representations of operations. The black arrows describe a direct flow of information. The dotted arrow between the model and the verification property indicates that verification properties are derived from the model. Each dotted outline contains the data and operations required for the corresponding step as mentioned in the description of the general automata-theoretic approach.

![Figure 1: Overview of the automata-theoretic approach, with the corresponding steps marked as numbered dotted outlines](image)

2.1 Conventional approach

In conventional model checking this approach is executed once and each step is finished before going to the next step.

The sequential execution of the above mentioned steps lead to the following two issues:

1. If Model M describes an infinite state space step 2 never stops in theory, because termination is not guaranteed. In practice however, the state space is generated until the memory limit is met; consequently this obtained state space is verified. However the problem then lies that it is not clear which part of the state space is verified.
2. In the situation that the state space generation does terminate, the complete product is produced before the verification is done.

These two issues are individually resolved by using bounded model checking for the first and on-the-fly model checking for the second issue. This is explained in the following subsections.

2.2 Bounded model checking

Bounded model checking defines a bound which is taken into account when the state space is generated during step 2. In bounded model checking this bound is defined by a constraint on the state space generation in such a way that a finite state space is generated. This implies that step 2 is proven to terminate. The definition of the bound determines to which extent the state space is generated and verified. Therefore bounded model checking can be regarded as an exploration strategy.

By setting an initial bound, it is determined what part of the complete state space is explored and subsequently is verified. In the absence of a counter example this bound is increased and in the next iteration a larger state space is generated and verified. The degree by which the bound is increased is the step size. Given an initial bound and a step size the bound can be increased until the state space becomes unmanageable or a predefined upper bound is set. The last successful bound verification is called the completeness threshold, and is the maximum bound to which the absence of the verification property is proven.

One example definition of a bound is the maximum path length from the initial state to which the state space is being generated. The step size in this case would be the increase of the maximum path length by $x$, the length added to the maximum path length of each iteration. This is depicted in figure 2, where the initial bound is set by $k$ and a step size of 1. The initial state space generated with length $k$ is coloured black, whereas the state space increase covered in the subsequent iteration with bound $k + 1$ is coloured grey.

![Figure 2: Bounded model checking with bound defined by path length, initial bound k and step size 1](image)

The bound definition is not necessarily defined by properties that are attributed to the state space (such as path length in the example). The bound can also be defined using properties and operations used by the model or used to describe the model [5].

The quality of bounded model checking is that this approach can guarantee the absence of a counter example within the completeness threshold. Because the bound is set, the otherwise possibly infinite state space becomes bounded and termination of the state exploration can be guaranteed [3]. Bounded model checking together with a step size offers a way to systematically analyze (in)finite state spaces by incrementally increasing the bound.

2.3 On-the-fly model checking

As mentioned in the introduction, on-the-fly model checking tries to prevent the complete generation of the state space. Instead of first generating the complete state space of the system and then checking it against the validation property, the properties are verified while exploring the state space [6].
When using on-the-fly model checking, the verification fails as soon as exploration and validation of a new state do not satisfy the given property. This means that the generation of the state space, producing the product automaton and doing the emptiness check (step 3, 4 and 5 of the above mentioned approach) are interleaved step-by-step. In figure 1 on-the-fly model checking can be regarded as the area covered by the areas 3, 4 and 5.

When the verification against the property fails, the path that led to this newly explored state is the faulty trace that violates the property. If a faulty trace is found, further exploration of the state space stops. Usually on-the-fly model checkers find a counter example without exploring the complete state space (assuming a counter example is present). Whenever a counter example is not present, the complete state space is explored and verified.

Breadth-first traversals are not well-suited to detect accepting runs in the automaton and therefore depth-first search-based algorithms are used for emptiness checks[7].

2.4 Combining bounded model checking with on-the-fly model checking

When bounded model checking is combined with on-the-fly model checking the question is raised how these steps as mentioned in the general approach should be interleaved. The following combined approach is derived:

Given a model and a verification property (step 1) the automaton of the negated property is derived (step 2). With an initial bound and a step size the state space generation is done per state (step 3). Each transition followed during state space generation (step 3) is used in producing the product automaton step by step (step 4). Each newly reached state that synchronizes with the verification automaton is checked on-the-fly if it violates the emptiness check (step 5). This is repeated until the bound is met or until a counter example is found.

3. PROBLEM STATEMENT

On-the-fly model checking algorithms are based on a depth-first search. This depth-first search is guaranteed to terminate because the input consequently obtained from step 3 is guaranteed to terminate by presence of the bound to which the state space is generated.

In absence of a counter example, the bound is increased. However, when the bound is increased the on-the-fly model check algorithm applied on the previously (finite) generated state space has already terminated. Therefore a new application of the on-the-fly model checking algorithm is needed on the continued generation of the product automaton P.

By reapplying the on-the-fly model checking algorithm every time the bound is increased the states that were already present in the previous application of the on-the-fly model checking algorithm are checked again. This is considered redundant work if states can be proven that they can not lead to counter examples.

When a state can be attributed such a characteristic, it is a so-called pocket state. States that are marked as such do not have to be considered by the emptiness check algorithm in future emptiness checks, because they have been proven not to be able to lead to counter example in a previous check. By using pocket states, the redundant work of rechecking these states can be reduced.

This leads to the following two questions:

1. How can it be determined that states are pocket states
2. Does the redundancy reduction by marking and using pocket states yield an increase in efficiency

4. APPROACH

The approach is structured as follows. First the requirements for marking pocket states are given and shown by using an example. Following this example one on-the-fly model checking algorithm is given: based on nested depth first search (NDFS) as proposed by Schwoon & Esparza in [6]. This algorithm is used to integrate the use of pocket states and the altered algorithm using pocket states is given.

4.1 Marking pocket states

The product automaton (subject to emptiness checking) is represented by a Büchi automaton. The original representation of the state space generated by the model is a Kripke[8] structure. In Kripke structures deadlock states are disallowed. To represent a deadlock state in a Kripke structure an edge is added from the deadlock state back to itself. Subsequently the Büchi product automaton may have states that have transitions to themselves due to translation from a Kripke structure to a Büchi automaton. However for the marking of pocket states, only the transitions matched by a transition in the model are considered.

To determine if a state is a pocket state the following requirements are to be met:

1. The state has no unexplored outgoing transitions
2. If the state has outgoing transitions other than to itself, these transitions lead to pocket states.

In order to meet the first requirement both the state in the generated state space and the corresponding state in the product automaton do not have unexplored outgoing transitions. Even if in the product automaton the state has no unexplored outgoing transitions, it can still be the case that in the state space the state has a transition crossing the current set bound. This is why additional information is needed from outside the product automaton P or this additional information needs to be represented in the product automaton.

This is illustrated in the product automaton P as depicted in figure [8]. The bound is given by path length and the step size is 1.

During the application of the emptiness check when bound = 0, S0 still has unexplored outgoing transitions that cross the bound. The same applies to S1 and S4 during bound = 1. During bound = 2, S2 and S3 are considered pocket states because they do not have outgoing transitions that cross the bound. As mentioned in section 2.3 the on-the-fly model checking emptiness checks are based on depth-first search. Therefore the information that S2 and S3 are marked as pocket states is stored and can immediately be used to determine that S1 has outgoing transitions other than those leading to S2 and S3 (which are pocket states). Subsequently S1 is marked as a pocket state. During following applications with bound > 2 any transitions leading to marked pocket states do need to be explored, because it has been proven in an earlier iteration that they can not lead to the detection of an accepting cycle.
4.2 Schwoon & Esparza’s algorithm

One on-the-fly model checking algorithm is by Schwoon & Esparza. This algorithm is based on a nested depth-first search algorithm (listed in listing 1) and is considered the best performing NDFS algorithm using single acceptance Büchi automata by Courverre in [9].

With the post operation all outgoing states reached by transitions that are matched by system transitions are obtained.

The colour codes have the following meaning:

white: initial colour for reachable states

cyan: a state of which the (blue or red) search has not yet been terminated

blue: a state of which the search has been terminated, and no previous states leading to this state include an accepting state

red: a state contained in a path by a search that contains an accepting state

The rationale behind the algorithm is that the blue search is the reachability analysis and the red search is the (nested) search detecting an accepting loop. The check on line 13-14 is for early detection of accepting loops. As soon as the blue search reaches a state where no deeper reachability analysis can be done, the algorithm colours this state blue and starts backtracking. It will move up to any previously explored states until it backtracks to a state that still has unexplored outgoing transitions and will continue exploring and colouring them.

4.3 Pocket states in Schwoon & Esparza

As mentioned before, the information about marked pocket states need to be stored outside the emptiness check algorithm. Also, additional information is required from outside the product automaton; whether or not the state being considered for marking as a pocket state has outgoing transitions that cross the bound.

Taking this into account, and including the requirements for marking pocket states as states in section 4.1, the alterations that led to Schwoon & Esparza’s altered algorithm presented in listing 2 and their corresponding line numbers are:

line 7-8: While in the red search, the algorithm attempts to find a cycle. A red search implies that an accepting state has been encountered. Trying to find cycles back to the accepting state through pocket states will not yield anything, for pocket states are proven not to lead to accepting states.

line 12: When doing reachability analysis it is considered redundant work to re-explore pocket states, for in a previous bound it has already been proven that this state does not lead to accepting states.

line 26: When the depth-first search has reached its current maximum depth, it starts backtracking. This is where every state needs to be checked if the corresponding state in the generated state space contains transitions that cross the bound (and are therefore not present in the product automaton).

line 27-30: If this is not the case, the state is a candidate for being marked as a pocket state. Subsequently all the states that can be directly reached following outgoing transitions are checked if these states are pocket states.
5. EXPERIMENTS

In order to carry out experiments a model checker is used that is a realization of the mentioned general automata-theoretic approach as mentioned in section 2. The measurements used and experiments done to determine if the use of pocket states yields an increase of efficiency.

5.1 Model checker

The models used in experiments are made using GROOVE [10] (version 2.0.0.3). This tool uses graphs to represent the states of the system. Outgoing transitions of states are defined by graph transformation rules. When a graph transformation rule can be applied to a graph, the graph(state) has an outgoing transition. By applying a graph transformation rule a new graph is produced, which represents the state reached by following this transition.

In addition to using transformation rules for generating state spaces, they can be used to determine if a rule is applicable on a state. This in turn can be used to represent propositions over these states. Verification properties are described in linear temporal logic (LTL) [11]). LTL uses the usual logic connectives ¬, ∨, ∧, → and temporal operators ∃ (next), ∅ (eventually/future), □ (always/globally), U (until) and R (release) to reason over propositions.

From this LTL formula an equivalent automaton can be obtained that is used to produce the product automaton. As a representation, Büchi automata are used to represent the negated verification property and is obtained by using LTL2Bua [12].

The bound definition for the bounded model checking is defined by placing a bound on the size of a graph. This is the maximum amount of nodes of which a graph can exist. This is a bound definition using structural properties of the graph transformation systems. The rationale behind this is that given a finite set of graph production rules (which are contained in the model and therefore finite) and a given bound to the amount of nodes that a produced graph may contain, it follows that the amount of uniquely producible graphs is finite. Thus this is a valid bound to limit the state space generation.

Checking the absence of loops containing accepting states in the product automaton P is done with Schwoon & Esparza’s algorithm which is adapted to mark pocket states. The used implementation of this altered algorithm was available in the benchmarking class of the tool.

5.2 Measurements

To obtain measurements the benchmarking class accompanied with the used GROOVE version is used. This benchmarking class uses the components as mentioned in the previous subsections and derives measurements by using the java System class.

An experiment consists of verifying a model against a certain property twice. Once using pocket states and again without the use of pocket states. For an experiment an initial bound, step size and upper bound is set using graph size, to which the state space is generated. The set upper bounds are set relatively low (maximum of 5 iterations) in order to obtain the size of the state space within the bounds of available memory.

During each experiment the following measurements are made:

- The iteration in which the experiment is terminated
- Time taken to execute the experiment
- Memory used during the execution of model checking
- Amount of pocket states found during pocket state enabled runs
- Number of states explored
- Number of transitions made

5.3 Experiments

Two models are subject to experiments. One is a model of an extendible buffer. The other is the ‘clique’ model, an alteration of the gossiping girls model.

This extendible buffer is a model of an extendible stack with an initial capacity of two. Initially the stack contains no items. Items can be added to the stack by using the put operation and removed
from stack by using the \textit{get} operation. In addition to these rules, the capacity of the stack can be increased by one when the stack is empty using the \textit{expand} operation.

The ‘clique’ model is an alteration on the gossiping girl model and is as follows. Two girls are forming a clique. Each girl has a secret that is unknown to other girls. Girls can share secrets with each other (\textit{swapSecrets}), exchanging one secret each time they speak. Additionally, new girls can join the clique (\textit{newGirl}). Each new girl brings in a unique secret. At any time when there are 4 or more girls in the clique they can decide that their clique is big enough (\textit{cliqueFull}), and the \textit{newGirl} rule is from thereon no longer applicable. Finally, whenever 3 or more girls share a secret, they can declare it a public secret (\textit{publicSecret}) and it is determined by the clique that it is a secret of which no longer should be talked. Consequently, this secret can no longer be exchanged.

Experiments, their accompanied LTL formulae and setup with regards to initial bound and step size are listed in table 1.

### Table 1: Experiments and their configuration, tests 1,2,3 using the Clique model, tests 4 using the extendible buffer

<table>
<thead>
<tr>
<th>Test</th>
<th>LTL formula</th>
<th>Bound</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((\text{swapSecrets} \rightarrow \Diamond \neg \text{publicSecret}))</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>((\text{newGirl} \rightarrow \Diamond \text{publicSecret}))</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>((\text{newGirl} \cup \Diamond \text{publicSecret}))</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>((\text{put} \rightarrow \Diamond \neg \text{get}))</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The experiments are run on a machine with a AMD XP 2000+ processor with 1 gB of memory using Java 1.6.0_02-b05 virtual machine.

### 6. RESULTS

The experiments yield the following results. The Tables 2-4 display the results of Clique model (as described in section 5.6.2). Table 5 displays the result of the tested Extendible buffer model (as described in section 5.6.1). All experiments have a counter example found, bar experiment 3, which is displayed in table 4. Because no counter example is found, the state space is fully explored each iteration resulting in a long duration before the verification stopped.

### Table 2: Clique model, checked against property: \(\Box(\text{swapSecrets} \rightarrow \Diamond \neg \text{publicSecret})\), counter example found in iteration 5

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Non-pocket</th>
<th>Pocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>1194</td>
<td>1189</td>
</tr>
<tr>
<td>space (kb)</td>
<td>391</td>
<td>391</td>
</tr>
<tr>
<td>nr. pocket states</td>
<td>n/a</td>
<td>0</td>
</tr>
<tr>
<td>states</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>transitions</td>
<td>85</td>
<td>85</td>
</tr>
</tbody>
</table>

These results show that a time efficiency increase is not measured when using pocket states. Comparison of the performance of using pocket states to not using them leads to the following observations:

- For experiments that yield counter examples in early iterations, hardly any performance difference is measured in either memory or time.
- Whenever marking of pocket states is enabled, more memory is used for the state space.
- When encountering only a small amount of pocket states during marking, the time in which the state space is checked is equal.

### Table 3: Clique model, checked against property: \(\Box(\text{newgirl} \rightarrow \Diamond \text{publicSecret}), \) counter example found in iteration 5

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Non-pocket</th>
<th>Pocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>1058</td>
<td>1052</td>
</tr>
<tr>
<td>space (kb)</td>
<td>312</td>
<td>321</td>
</tr>
<tr>
<td>nr. pocket states</td>
<td>n/a</td>
<td>2</td>
</tr>
<tr>
<td>states</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>transitions</td>
<td>41</td>
<td>41</td>
</tr>
</tbody>
</table>

### Table 4: (Clique mode, check against property: \(\Box(\text{newgirl} \rightarrow \Diamond \text{publicSecret}), \) no counter example, upper bound met in iteration 5

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Non-pocket</th>
<th>Pocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>197306</td>
<td>197352</td>
</tr>
<tr>
<td>space (kb)</td>
<td>54908</td>
<td>57352</td>
</tr>
<tr>
<td>nr. pocket states</td>
<td>n/a</td>
<td>19</td>
</tr>
<tr>
<td>states</td>
<td>10191</td>
<td>10191</td>
</tr>
<tr>
<td>transitions</td>
<td>67114</td>
<td>67114</td>
</tr>
</tbody>
</table>

### Table 5: Extendible buffer model, checked against property: \(\Box(\text{put} \rightarrow \Diamond \neg \text{get}), \) counter example found in iteration 4

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Non-pocket</th>
<th>Pocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>619</td>
<td>621</td>
</tr>
<tr>
<td>space (kb)</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>nr. pocket states</td>
<td>n/a</td>
<td>1</td>
</tr>
<tr>
<td>states</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>transitions</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

### 7. CONCLUSION AND DISCUSSION

Pocket states can be marked as such if the state has no unexplored outgoing transitions and all outgoing states that lead to states other than itself are marked as pocket states. This is under the premise that the transitions under consideration are matched by transitions modeled by the system and not those to represent deadlocked states in a Kripke structure, as described in section 4.1.

From the experimental results can be concluded that the reduction obtained from using pocket states does not always lead to an increase of efficiency. Using pocket states imposes additional memory use to maintain the information if a state is marked as a pocket state or not. This is best seen in table 4 of the results in section 6. However, the java System class is not the most accurate
with regards to measuring memory used, and the use of a stand-
alone profiler is recommended.

The experiments done used small models and simple verifica-
tion properties. The maximum amount of pocket states detected was
19. The use of more complex models and experiments with ver-
fications that use more iterations are suggested. Larger models
with more complex verification properties may result into more
marked more pocket states. This is subject to research.

It is believed that there is a relation between how many iterations
of the emptiness check are done together with in which iteration
pocket states are marked and the possible increase in efficiency.
Additionally groups of pocket states are believed to yield a larger
increase of efficiency than single pocket states. Whether and to
which extent these relations yield are subject to further research.

Acknowledgements
First of all I would like to thank Harmen Kastenberg for sparking
my interest and introducing me to this field of research. Lodewijk
Bergmans and Marielle Stoelinga for being cheerful and relaxed
teachers that gave valuable feedback during my journey on per-
fecting writing papers. And of course thanks to my peers in the
track ‘Building High-Quality Systems - With model checking and
Aspects’ for their feedback throughout the course.

REFERENCES
[1] E. M. Clarke, O. Grumberg, and D. A. Peled, Model

“Model-checking: A tutorial introduction,” in SAS ’99:
Proceedings of the 6th International Symposium on Static
Analysis. (London, UK), pp. 330–354, Springer-Verlag,
1999.

[3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu,

Nieuwsbrief van de Nederlandse Vereniging voor

[5] A. Rensink, “Time and space issues in the generation of
graph transition systems,” in International Workshop on
Graph-Based Tools (GraBaTs), Electronic Notes in
Theoretical Computer Science. Elsevier Science

verification algorithms..” in TACAS 2005, pp. 174–190,
2005.


the Construction and Analysis of Systems. 13th
International Conference, TACAS 2007, Held as Part of the
Joint European Conferences on Theory and Practice of
Software. ETAPS 2007 Braga, Portugal, March 24 - April
1, 2007, Proceedings, vol. 4424 of Lecture Notes in

“On-the-fly emptiness checks for generalized Büchi
automata,” in Proceedings of the 12th International SPIN
Workshop on Model Checking of Software (P. Godefroid, 
ed.), vol. 3639 of Lecture Notes in Computer Science,

space generation,” in Applications of Graph
Transformations with Industrial Relevance (AGTIVE),
vol. 3062 of Lecture Notes in Computer Science,

Proceedings of the 18th IEEE Symposium on the
Foundations of Computer Science, pp. 46–57, IEEE

automata,” Z. Math. Logik und Grundl. Math., vol. 6,

translation,” in Proceedings of the 13th International
Conference on Computer Aided Verification (CAV’01)
(G. Berry, H. Comon, and A. Finkel, eds.), vol. 2102 of
Lecture Notes in Computer Science, (Paris, France),