Breadth-Bounded Model Checking

Maarten G. Meulen, Frank P.M. Stappers, and Tim A.C. Willemse

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. Model checking large concurrent systems is a difficult task, due to the infamous state space explosion problem. To combat this problem, a technique called Bounded Model Checking has been proposed. This techniques relies on restricting the level of unfoldings of the transition relation of a given specification. This technique is quite effective for verifying requirements that are relatively close to the initial state of the system's behaviour. Unfortunately, this technique is not adequate for disproving requirements which occur at levels that are relatively deep within the system. In this paper, we study an alternative approach to BMC by restricting the breadth of the transition relation, based on a *Highway simulation*. This allows us to find violations to (1) properties that lurk deep in a specification, and (2) properties that require lengthy counterexamples. Our experiments show that the method is complementary to BMC, and is effective in many practical applications.

1 Introduction

Designing software for devices is a complex and challenging task. Analysing designs prior to actually implementing these is good practice, though still not very commonplace. Model checking is a popular technique for analysing designs, and the functioning of a device as a whole. This technique essentially relies on an exhaustive exploration of the system's behaviour; it uses *temporal logic* for formalising requirements on the system's behaviour. Model checking can be used to prove that requirements hold, or show that they are violated. However, traditional model checking often suffers from the infamous state space explosion, rendering the technique useless.

Various authors have proposed techniques to combat the state space explosion, based on *partial* explorations of the state space. A notable example is the technique of *Bounded Model Checking* (BMC), pioneered by Biere *et al* [4, 6]. This technique has been demonstrated to be quite effective in the verification of industrial designs. Even though BMC is incomplete (it does not guarantee a true or false determination for all specifications [6]), it has been used to find bugs in situations where more traditional verification techniques fail completely. Being incomplete still allows BMC to be used effectively for *bug-hunting*, *i.e.*, finding counterexamples to conjectured liveness or safety requirements.

BMC basically works like a breadth-first exploration of the state space, where the (symbolic) transition relation is unwound up-to a predetermined bound under the rationale that bugs appear even in the partially unfolded parts of the state space. Rather than building a concrete representation of the state space, a propositional formula is constructed representing the unfolded transition system. A temporal logic formula is encoded on top of the propositional formula, and the entire formula is subsequently handed over to a SAT solver. Much of the research in BMC is therefore dedicated to finding more efficient and effective propositional encodings for various classes of logics, e.g., for ACTL [20, 22] and $\Box L_\mu$, the universal fragment of the μ-calculus [19].

The breadth-first exploration underlying BMC is also a weakness. The effectiveness of BMC is sensitive to the branching degree of the specification and depends on the depth at which the property that is being checked occurs. High branching degrees more quickly lead to a state explosion at shallower depths, meaning that finding errors for *deep* and *long-running* properties is harder. This is testified in e.g. [19].

In this paper, we take an alternative approach to bug-hunting, which we dub *Breadth-Bounded Model Checking* (BBMC). Rather than restricting the number of unfoldings of the transition relation, BBMC restricts the breadth of the exploration. In essence, we construct a homogeneous "slice" of the state space, that we subsequently subject to verification using standard tooling. The slice is constructed using algorithms that are based on a *Highway simulation* [10], which is modified for the purpose of subsequent verification. We show that our algorithms construct state spaces that are *simulated* by the full state space of the symbolic input specification. The results by Loiseaux *et al* [17], regarding the reflection of $\Diamond L_\mu$, the existential fragment of the μ-calculus (or, dually, the *preservation* of $\Box L_\mu$), subsequently guarantees that any witness to an $\Diamond L_\mu$ formula, found in our state space is also in the original specification. This can be used to find counterexamples for typical safety and liveness properties expressed in $\Box L_\mu$.

In order to obtain a good coverage of the inspected state space, BBMC employs a uniform randomisation to construct a state space. A positive effect of the randomisation is that successive runs increase the level of confidence in the correctness of the input specification. We test the practical effectiveness of BBMC, by subjecting the technique to several case studies. These vary in size and complexity. The effectiveness is measured by offsetting the number of states BBMC produces to the number of states BMC requires. The picture that emerges from these experiments confirms our basic intuition: BMC generally performs well on shallow properties, whereas BBMC performs better for properties at a deeper level and properties that require lengthier counterexamples. This essentially means that BBMC and BMC are complementary.

This paper is structured as follows. In Section 2, we introduce the basic semantic framework on which BBMC relies. Section 3 subsequently introduces our state space exploration algorithms underlying BBMC. The experiments that we

conducted using BBMC and BMC are described in Section 4. Section 5 discusses related work, and conclusions and future work are presented in Section 6.

2 Preliminaries

We assume that systems are described by state transition systems. In particular, we use the notion of *Kripke Structures* to outline our theory; the use of this particular framework is inconsequential to our theory. Therefore all results transfer immediately to other models such as *Labelled Transition Systems*.

A Kripke Structure is a directed graph in which the nodes are labelled with sets of atomic propositions; for simplicity, we assume these are taken from a fixed, finite but non-empty set of atomic propositions names denoted AP.

Definition 1. *A* Kripke Structure *is a quadruple* $\mathcal{K} = \langle S, \bar{s}, L, \rightarrow \rangle$*, where:*

- S *is a finite set of* states*;* $\bar{s} \in S$ *is the* initial state*;*
- $L{:}S \rightarrow 2^{AP}$ *is the* proposition labelling*;*
- $\rightarrow \subseteq S \times S$ *is the* transition relation*; an element* $(s,t) \in \rightarrow$ *is called a* transition *and is commonly written as* $s \rightarrow t$*.*

We say the Kripke Structure is total if the transition relation $\rightarrow$ *is a total relation, i.e., if for all* $s \in S$*, there is a* $t \in S$*, such that* $s \rightarrow t$*.*

Let $\mathcal{K} = \langle S, \bar{s}, L, \rightarrow \rangle$ be a Kripke Structure. The set of *successor states* of a state $s \in S$, denoted by $\mathsf{succ}(s)$, is the set $\{t \in S \mid s \rightarrow t\}$. We generalise this notation to sets of states as follows: $\mathsf{succ}(Q) = \bigcup_{s \in Q} \mathsf{succ}(s)$. A state $s \in S$ is a *sink state* (also known as a *deadlock* state), denoted $\dagger s$ iff $\mathsf{succ}(s) = \emptyset$.

The behaviours of a Kripke Structure are represented by the *execution paths*, hereafter referred to as *paths*, of the Kripke Structure. A *path* in $\mathcal{K}$ is either a finite sequence $\langle s_0, s_1, s_2, \ldots, s_n \rangle$ of states satisfying $\dagger s_n$ and for all $i < n$, $s_i \rightarrow s_{i+1}$; or it is an infinite sequence $\langle s_0, s_1, s_2, \ldots \rangle$ of states such that for each $i \in \mathbb{N}$, we have $s_i \rightarrow s_{i+1}$. If π is a finite or infinite path $\langle s_0, s_1, s_2, \ldots \rangle$, then $\pi(i)$ denotes state s_i. We say that a state $s \in S$ in $\mathcal{K}$ is *reachable* iff there is a path π with $\pi(0) = \bar{s}$ and for some i, we have $\pi(i) = s$. The restriction of $\mathcal{K}$ with respect to the reachable states, denoted $\mathcal{K}_{\mathsf{reach}}$, is defined as the quadruple $\langle S_{\mathsf{reach}}, \bar{s}, L \rceil S_{\mathsf{reach}}, \rightarrow \cap S^2_{\mathsf{reach}} \rangle$, where:

- $S_{\mathsf{reach}} = \{s \in S \mid s \text{ is reachable}\}$;
- for every $T \subseteq S$, $L \rceil T = \lambda s \in T.\ L(s)$;

A large number of behavioural equivalences and pre-orders for Kripke Structures and similar models have been proposed in the literature. A prominent pre-order is the *simulation pre-order* [5]. It is a natural pre-order for matching an implementation with a specification when preservation of the branching structure is of importance. Simulation pre-orders can be used to combat the state explosion problem that inhibits model checking, by minimising the state space of a given system modulo simulation pre-order, prior to model checking. Formally, a simulation relation between two Kripke Structures is defined as follows:

Definition 2. *Let* $\mathcal{I} = \langle S, \bar{s}, L, \rightarrow \rangle$ *and* $\mathcal{S} = \langle S', \bar{s}', L', \rightarrow' \rangle$ *two Kripke Structures, the* implementation *and* specification. *A relation* $\mathcal{R} \subseteq S \times S'$ *is said to be a* simulation *relation iff, whenever* $s \; \mathcal{R} \; s'$*, then:*

1. $L(s) = L'(s')$*;*
2. *if* $s \rightarrow t$*, then for some* t'*,* $s' \rightarrow' t'$ *and* $t \; \mathcal{R} \; t'$*.*

We say that $\mathcal{S}$ simulates $\mathcal{I}$*, denoted* $\mathcal{I} \sqsubseteq \mathcal{S}$ *iff there is some simulation relation* $\mathcal{R}$*, such that* $\bar{s} \; \mathcal{R} \; \bar{s}'$*.*

An important property of the simulation pre-order is that it reflects the existential fragment $\Diamond L_\mu$ of the modal μ-calculus [17], i.e., the fragment of μ-calculus formulae containing only the modality $\Diamond$, having no negations or implications. Indeed, Loiseaux *et al* [17] showed the following theorem:

Theorem 1. *Let* $\mathcal{I}$ *and* $\mathcal{S}$ *be two given Kripke Structures. If* $\mathcal{I} \sqsubseteq \mathcal{S}$*, then for every formula* $\phi \in \Diamond L_\mu$*: if* $\mathcal{I} \models \phi$ *then* $\mathcal{S} \models \phi$*, where* $\models$ *is the standard satisfaction relation.* □

Note that the modal μ-calculus is strictly more expressive than the more user-friendly logic CTL^* [5], and its sublogics LTL and CTL; moreover, $\Diamond L_\mu$ is strictly more expressive than the existential fragments ECTL^*, ECTL and existential LTL. As a result, the above theorem also holds for the latter fragments.

Note that the dual of Theorem 1, i.e., the preservation of $\Box L_\mu$, is often used in conjunction with abstract interpretation for proving safety properties. In this paper, we rely on the reflection of $\Diamond L_\mu$ for *bug-hunting*: we search for violations of safety and liveness properties in a simulation of a given specification.

3 Breadth-Bounded Model Checking

A bottleneck in a full state space exploration is that the state space may simply be too large to explore given a certain amount of time: to all intents and purposes, it might as well be infinite. We address this problem by applying model checking using a state space that is generated using a randomised, bounded-breadth-first exploration, based on a so-named *Highway simulation* [10]. We show that the resulting state space can be used as a basis for model checking the existential fragment of the μ-calculus. We first repeat the basic Highway simulation algorithm, after which we describe the modifications that are required to turn this algorithm into a partial state space exploration algorithm, suitable for model checking.

3.1 Highway Simulation

Highway simulation traverses a symbolic state space (for the sake of simplicity represented by an explicit Kripke Structure in all our algorithms) in a homogeneous fashion. It simultaneously explores several *lanes*, consisting of a sequence of states. A predefined number of successor states for the last-visited states in

the lanes are chosen by randomly selecting these among all successor states that have not yet been visited in some previous lane, see Algorithm 1. In this algorithm, V denotes the set of *visited states* and Q_d, the so-called *queue*, is the set of states, last-visited by the lanes. The variables V and Q_i are related as follows: in each iteration, $V = \bigcup_{i \leq d} Q_i$; moreover, we have $Q_i \cap Q_j = \emptyset$ for all $i \neq j$. The entire algorithm is parameterised by a Kripke Structure $\mathcal{S}$ and a breadth-bound $N > 0$; the latter is used to cap the size of each queue. Note that the assignment in line 4 can be approximated efficiently by following the techniques outlined in [10]; here, we stick to the current presentation for ease of reasoning.

Definition 3. *A subset $T \subseteq S$ is called* N-close *to S iff $|T| = \min(|S|, N)$.*

Thus, an N-closed subset of a set of states S has N elements at most, and at least N elements if $|S| > N$.

Algorithm 1 The Highway Simulation Algorithm

Require:
Breadth bound $N > 0$;
Kripke Structure $\mathcal{S} = \langle S, \bar{s}, L, \rightarrow \rangle$

1: $V, d, Q_0 := \{\bar{s}\}, 0, \{\bar{s}\}$;
2: **while** $Q_d \neq \emptyset$ **do**
3: $\quad R := \mathsf{succ}(Q_d) \setminus V$;
4: $\quad \mathcal{Q} := \{Q \subseteq R \mid Q \text{ is } N\text{-close to } R\}$;
5: $\quad$ **Randomly choose** Q_{d+1} **from** $\mathcal{Q}$;
6: $\quad V, d := V \cup Q_{d+1}, d + 1$;
7: **end while**

The Highway simulation algorithm terminates at some depth D whenever the set of lanes can no longer continue to explore new places, i.e., when every successor state of the states in queue Q_D leads back to a state contained in some prior queue Q_i ($i \leq D$). In particular, this entails that a 1-bounded highway simulation behaves like a random walk, with the exceptions that it tries to avoid revisiting states and it terminates if this can no longer be done. It is not too hard to see that the algorithm always terminates for $\mathcal{S}$.

A visualisation of a typical 2-bounded highway simulation of a state space is depicted in Fig. 1. In this picture, the small, shaded ovals indicate the queues Q_j (for some j), whereas the big ovals represent the set of successors of a previous queue, *i.e.*, $\mathsf{succ}(Q_{j-1})$. The visited and considered states are depicted by small circles inside the ovals; a solid circle indicates that the state has been considered in the construction of some prior queue (but was not chosen at that time), whereas an open circle represents a state that was never considered in the construction of a previous queue. For instance, in Fig. 1, one of the solid circles in Q_{i+3} could be (but obviously need not be) the state in $\mathsf{succ}(Q_{i+1}) \setminus Q_{i+2}$.

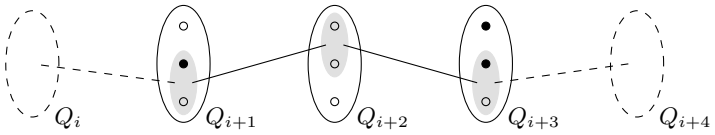


Fig. 1: Example of a Highway Search

3.2 Partial State Space Construction

A Highway simulation does not record transitions; its main purpose is to visit a series of states, with the intention to check for state properties. Indeed, in [10], Highway simulation was used exclusively for *searching*, leading to a Highway search technique.

Our aim is to modify Highway simulation in such a way, that it retains as much of the original state space as is viable without violating the basic idea of a Highway simulation. This is achieved by reconstructing as many transitions as possible. Adding all transitions can be done at the end of a Highway simulation, see also the last line in Algorithm 2. For the purpose of *on-the-fly* model checking of fragments of $\Diamond L_\mu$, such as existential LTL, we analyse the types of transitions that are considered in some more detail.

We observe that in a Highway simulation, between any two queues Q_i and Q_{i+1}, the set Q_{i+1} consists entirely of successors of states in Q_i. Furthermore, some successors of Q_i are not considered at all since these have already appeared in some Q_j ($j \leq i$). Lastly, there can be states in some queue Q_{i+k}, for $k > 1$, that have been considered as successors for Q_i, but which were not chosen at that point (see also the solid states in Fig. 1). Summarising, we distinguish three types of transitions:

1. *single-steps:* transitions from states in Q_i to Q_{i+1};
2. *back-edges:* transitions from states in Q_i to some Q_j ($j \leq i$);
3. *jumps:* transitions from states in Q_i to Q_{i+j} ($j > 1$).

Using the above break-down, the assignment in the last line in Algorithm 2 can be replaced by an assignment adding single-step transitions and back-edges in the inner loop of Algorithm 2 at no additional computation costs. Jumps are still added *a posteriori*, i.e., upon termination of the inner loop.

Fig. 2 illustrates the various types of transitions that can be considered. Solid lines represent the single-step transitions; dashed lines represent *jumps*; dotted lines represent *back-edges*. Again, solid states indicate that the state has been considered for election at some prior moment.

Another deviation from the original Highway simulation algorithm is that we disallow the state space construction to introduce non-specified sink states. We do so because we deem checking for reachability of sink states to be part of a set of basic verifications conducted for every system. Moreover, ensuring totality allows one to reuse the standard semantics of most temporal logics.

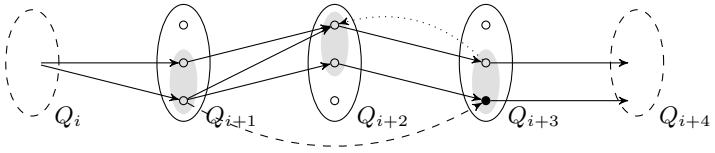


Fig. 2: Snapshot of a Highway Simulation recording transitions.

Algorithm 2 Bounded Breadth-First Exploration

Require:
Breadth bound $N > 0$;
Kripke Structure $\mathcal{S} = \langle S, \bar{s}, L, \rightarrow \rangle$

Ensure:
$\mathcal{H} = \langle V, \bar{s}, L\rceil V, \rightsquigarrow \rangle$

1: $V, d, Q_0, := \{\bar{s}\}, 0, \{\bar{s}\}$;
2: **while** $Q_d \neq \emptyset$ **do**
3: $\quad R := \mathsf{succ}(Q_d) \setminus V$;
4: $\quad \mathcal{Q} := \{Q \subseteq R \mid Q \text{ is } N\text{-close to } R \text{ and } \forall s \in Q_d : \dagger s \vee \exists t \in Q \cup V : \ s \rightarrow t\}$;
5: $\quad$ **Randomly choose** Q_{d+1} **from** $\mathcal{Q}$
6: $\quad V := V \cup Q_{d+1}$;
7: $\quad d := d + 1$;
8: **end while**
9: $\rightsquigarrow := V^2 \cap \rightarrow$;

Preservation of totality is achieved by the following modification to the way a Highway simulation constructs its next queue: rather than considering an arbitrary N-close subset of $\mathsf{succ}(Q_d)$, we require that the subset is chosen such that every non-sink state in Q_d has *at least one* successor state in some queue Q_j with $j \leq d + 1$. This essentially ensures progress of every lane. The next proposition formalises that Algorithm 2 indeed preserves totality of the transition relation of the input specification.

Proposition 1. *For all $N > 0$, and total Kripke Structures $\mathcal{S}$, Algorithm 2 yields a total Kripke Structure $\mathcal{H} = \langle V, \bar{s}, L\rceil V, \rightsquigarrow \rangle$ upon termination.*

Proof. Obviously, $\mathcal{H}$ is a Kripke Structure, so it suffices to show that it is a total Kripke Structure. Assume that $\mathcal{S}$ is a total Kripke Structure, i.e., the transition relation $\rightarrow$ is total. Upon termination of the algorithm, we have $\rightsquigarrow = V^2 \cap \rightarrow$. Since $\rightarrow$ is total, it suffices to show that for every $s \in V$, there is a state $t \in V$, such that $s \rightarrow t$. Let $s \in V$. By the invariant $V = \bigcup_{0 \leq i} Q_i$, we find that there is a queue $Q_j \subseteq V$ such that $s \in Q_j$. By line 4, the queue Q_{j+1} satisfies that $s \rightarrow t$ for some $t \in Q_{j+1}$ if not $s \rightarrow t$ for $t \in \bigcup_{k \leq j} Q_k$. Hence, also $t \in V$. Thus, the transition relation $\rightsquigarrow$ is total. □

Note that it is possible for the input to be a non-total Kripke Structure, whereas the output is still a total Kripke Structure. This is due to the randomness in the algorithm, and the fact that the algorithm does not introduce sink states by itself. The input and output Kripke Structures are related by the below theorem.

Theorem 2. *For all $N > 0$, and all Kripke Structures $\mathcal{S}$, Algorithm 2 yields a Kripke Structure $\mathcal{H}$ satisfying $\mathcal{H} \sqsubseteq \mathcal{S}$.*

Proof. Let $N > 0$, and assume $\mathcal{S} = \langle S, \bar{s}, L, \rightarrow \rangle$ is an arbitrary Kripke Structure. Let $\mathcal{H} = \langle V, \bar{s}, L\rceil V, \rightsquigarrow \rangle$ be the output of Algorithm 2 upon inputs $\mathcal{S}$ and N. Observe that upon termination of the algorithm we have $\{\bar{s}\} \subseteq V \subseteq S$ and $\rightsquigarrow \subseteq \rightarrow$; $\mathcal{H} \sqsubseteq \mathcal{S}$ then holds because the identity relation $\mathcal{I}_{=} = \{(s, s) \in S \times S \mid s \in V\}$ is a simulation relation. The latter follows immediately. □

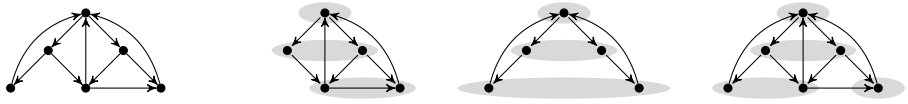
The purport of the above theorem is that the validity of a $\Diamond L_\mu$ formula on the Kripke Structure $\mathcal{H}$ carries over to the Kripke Structure $\mathcal{S}$:

Corollary 1. *Let $N > 0$ and assume $\mathcal{S}$ is an arbitrary Kripke Structure. Let $\mathcal{H}$ be the output of Algorithm 2 upon inputs $N > 0$ and Kripke Structure $\mathcal{S}$. Then for any formula $\phi \in \Diamond L_\mu$, we have $\mathcal{H} \models \phi$ implies $\mathcal{S} \models \phi$.* □

The corollary follows immediately from the combination of Theorem 1 and Theorem 2, and is essentially the basis for our *Breadth-Bounded Model Checking* methodology.

Note that the power of the full μ-calculus can be utilised if the state space that is constructed by Algorithm 2 yields the set of all reachable states of the input specification $\mathcal{S}$. The equivalence $\mathcal{S}_{\mathsf{reach}} = \mathcal{H}$ follows immediately from the condition $\rightsquigarrow = \rightarrow \cap (V \times S)$, which can be checked efficiently while constructing the transition relation $\rightsquigarrow$. Observe that for sufficiently large N, Algorithm 2 behaves as a breadth-first exploration. It can be the case that for some bound N, $\mathcal{S}$ is reconstructed using more iterations than a straightforward breadth-first exploration would require, as witnessed by the following example.

Example 1. Let $\mathcal{S}$ be the Kripke Structure as depicted below (left). Running Algorithm 2 on $\mathcal{S}$ with bound $N = 2$ can lead to the three Kripke Structures depicted next to it; the queues as explored by the algorithm, in the order top to bottom, left to right, are represented by shaded areas.



Note that with bound $N = 3$, the algorithm behaves exactly as a breadth-first exploration, and gives rise to three queues (exactly the three levels of the breadth-first exploration). Furthermore, note that the rightmost Kripke Structure is isomorphic to $\mathcal{S}$, while the other two are not; these are merely simulated by $\mathcal{S}$. The rightmost Kripke Structure illustrates that a full exploration of $\mathcal{S}$ can be achieved with a bound smaller than the bound required to behave exactly as a breadth-first exploration. Moreover, it illustrates that in this case, more iterations than a breadth-first exploration are required to obtain $\mathcal{S}$, viz. 4 vs. 3. □

3.3 Graceful degradation

In general, Algorithm 2 yields smaller state spaces than that of the input specification. However these can still be extremely large. Moreover, if $\mathcal{S}$ is well-connected, Algorithm 2 will construct excessively large Kripke Structures, approaching the size of the original specification. This can be undesirable at times. In this section, we investigate a heuristic that can quicken termination, and in general leads to smaller state spaces.

The heuristic we describe takes inspiration from standard Bounded Model Checking techniques. The latter relies on a depth-bound for terminating the unfolding of a given state space. While we still wish to impose a depth-bound on our exploration, we cannot do so naively without running the risk of introducing sink states that were not present in the original specification.

Instead, we describe a graceful degradation mode of our algorithm. After reaching a given depth, the exploration of new states is only done so long as non-totality cannot be avoided. This is effectively achieved by gradually decreasing the queue width to 0, see Algorithm 3.

Algorithm 3 Bounded Breadth-First Exploration with Graceful Degradation

Require:
Graceful degradation depth $D > 0$;
Breadth bound $N > 0$;
Kripke Structure $\mathcal{S} = \langle S, \bar{s}, L, \rightarrow \rangle$
Ensure:
$\mathcal{H} = \langle V, \bar{s}, L\rceil V, \rightsquigarrow \rangle$

$V, d, Q_0 := \{\bar{s}\}, 0, \{\bar{s}\}$;
while $Q_d \neq \emptyset$ **do**
  if $d < D$ **then**
    $R := \mathsf{succ}(Q_d) \setminus V$;
    $\mathcal{Q} := \{Q \subseteq R \mid Q \text{ is } N\text{-close to } R \text{ and } \forall s \in Q_d\colon \dagger s \vee \exists t \in Q \cup V\colon s \rightarrow t\}$;
  else
    $R := \{s \in Q_d \mid \mathsf{succ}(s) \cap V = \emptyset \text{ and not } \dagger s\}$
    $\mathcal{Q} := \{Q \subseteq \mathsf{succ}(R) \mid Q \text{ is } |R|\text{-close to } \mathsf{succ}(R) \text{ and } \forall s \in R\colon \exists t \in Q\colon s \rightarrow t\}$;
  end if
  Randomly choose Q_{d+1} **from** $\mathcal{Q}$
  $V := V \cup Q_{d+1}$;
  $d := d + 1$;
end while
$\rightsquigarrow := V^2 \cap \rightarrow$;

Proposition 2. *For all $N > 0$, $D > 0$ and total Kripke Structures $\mathcal{S}$, Algorithm 3 yields a total Kripke Structure $\mathcal{H} = \langle V, \bar{s}, L\rceil V, \rightsquigarrow \rangle$ upon termination.*

Proof. Assume that $\mathcal{S}$ is a total Kripke Structure. Note that, due to Proposition 1, it suffices to prove that the sets Q_i that are selected from the set $\mathcal{Q}$,

constructed in line 8, does not break totality. Let $s \in Q_i$, for some $i \geq D$. We distinguish two cases:

1. $s \in R$. This means that Q_{i+1} is selected such that there is a state $t \in Q_{i+1}$ satisfying $s \rightarrow t$.
2. $s \notin R$. This means that $\mathsf{succ}(s) \cap V \neq \emptyset$ or $\dagger s$. The latter cannot be because of totality of $\mathcal{S}$. From the former, it follows that there is a state $t \in V$, satisfying $s \rightarrow t$.

Both cases lead to the observation that for some $t \in V$ we have $s \rightarrow t$. Hence, graceful degradation does not compromise totality. □

Theorem 3. *For all $N > 0$, $D > 0$ and arbitrary Kripke Structures $\mathcal{S}$, Algorithm 3 yields a Kripke Structure $\mathcal{H}$ satisfying $\mathcal{H} \sqsubseteq \mathcal{S}$.*

Proof. Identical to the proof of Theorem 2. □

4 Experiments

In this section, we put the *effectiveness* of our approach to the test. For this, we use a prototype implementation of our two Algorithms of Section 3. The first set of experiments illustrates our hypothesis that our methods can be very effective in systems with "deep" flaws. Our second batch of experiments deals with complex systems that have larger state spaces, and are intended to illustrate the effectiveness of our methods in practice. Note that in our experiments, we do not focus on time performance, the reasons being that (1) we did not optimise our prototype tool in any way, and, (2) for our comparisons with standard BMC, we used a BMC implementation based on explicit state model checking techniques, rather than the more common, possibly more time-efficient SAT based methods used in the literature [4, 6]; any comparisons of timing between the two approaches would therefore be skewed and meaningless. For the purpose of repeatability, we first describe the general setup of our experiments.

4.1 General Test Setup

Our prototype tool implementing Algorithms 2 and 3, is developed within the mCRL2 tool suite[1]. Contrary to our exposition in Section 2 and 3, the tool suite is *action based*, but, as mentioned earlier, the differences between Kripke Structures and Labelled Transition Systems are inconsequential for our theory. The state spaces generated by our prototype implementation are fed to the model checker of the tool suite, which checks (first order) modal μ-calculus formulae. For our standard BMC experiments, we used a standard breadth-first search with a threshold on the number of states that it is allowed to explore. It is important to note that this way, we obtain a very optimistic under-approximation

[1] the tool suite can be obtained from http://www.mcrl2.org, revision 5731. Our prototype tool can be obtained from http://www.win.tue.nl/∼fstapper

on the number of states required for the verification using standard BMC, as the unfolding of the state space can now be stopped in between two exploration levels.

The experiments were run concurrently on several contemporary dual and quad-core PCs running recent distributions of Linux (FC8, SUSE 11), with varying configurations. Typical running times of our state space constructions were in the order of seconds; memory consumption was marginal in most cases, never exceeding 100 Mb. Since our method is probabilistic, each measurement was repeated 100 times, and our reported figures are averages.

4.2 Experiment I

We test the effect of increasing branching degrees on the capabilities of our approach to detect 'flaws'; we compare against the standard BMC method. Our hypothesis is that, as the branching degree increases flaws that can be found deeper in a specification are easier to detect using our methods, whereas flaws occurring at shallower depths are easier to detect using BMC.

Our experiments use two well-known data communications protocols, viz. the *Alternating Bit Protocol* (ABP) and the *Concurrent Alternating Bit Protocol*, see *e.g.* [18], a slightly more complex version of the ABP, in which more communications ports can be active simultaneously. The branching degree of the protocols is controlled by increasing the (finite) set of messages M that can be communicated. We varied the size of the set M from 2,4,8 to 16. The effect this has on average fanout and the number of states in each protocol is listed in the table below:

		Number of States / Average fanout			
Model	Levels	$\|M\| = 2$	$\|M\| = 4$	$\|M\| = 8$	$\|M\| = 16$
ABP	20	74 / 1.24	146 / 1.26	290 / 1.26	578 / 1.27
CABP	26	464 / 3.52	1,040 / 3.66	2,576 / 3.84	7,184 / 4.03

The properties we check using BBMC and BMC are the following:

I there is an infinite path in which a read message is never delivered;
II it is possible to consecutively receive and send two different messages.

Both properties hold for both protocols. We ran our prototype implementation of Algorithm 2 on both protocols and both properties, varying the breadth bound N from 1 up to, and including 6. Table 1 below indicates the probability of actually detecting a witness to the properties in the resulting state spaces for each breadth bound; as the variation is insignificant between the four different instances of each protocol, we give margins for these probabilities. Fig. 3 visualises the results of our experiments on the ABP and the CABP for the two properties. It displays the average number of states constructed by our algorithm, normalised against the number of states needed for the BMC routine to find a witness. Thus, all plots above the 1.0 lines indicate that BMC is more efficient, whereas all plots below the 1.0 line indicate that BBMC produces fewer states.

		Success Rate (%) per Breadth Bound					
Model	Property	1	2	3	4	5	6
ABP	I	52–69	80–91	91–100	97–100	99–100	99–100
	II	7–10	16–35	49–59	66–72	75–100	85–100
CABP	I	100	100	100	100	100	100
	II	35–52	91–94	96–100	99–100	99–100	100

Table 1: Probability of finding a witness to properties I and II using BBMC. The ranges are due to the variations in the probability of finding a witness for different breadth bounds.

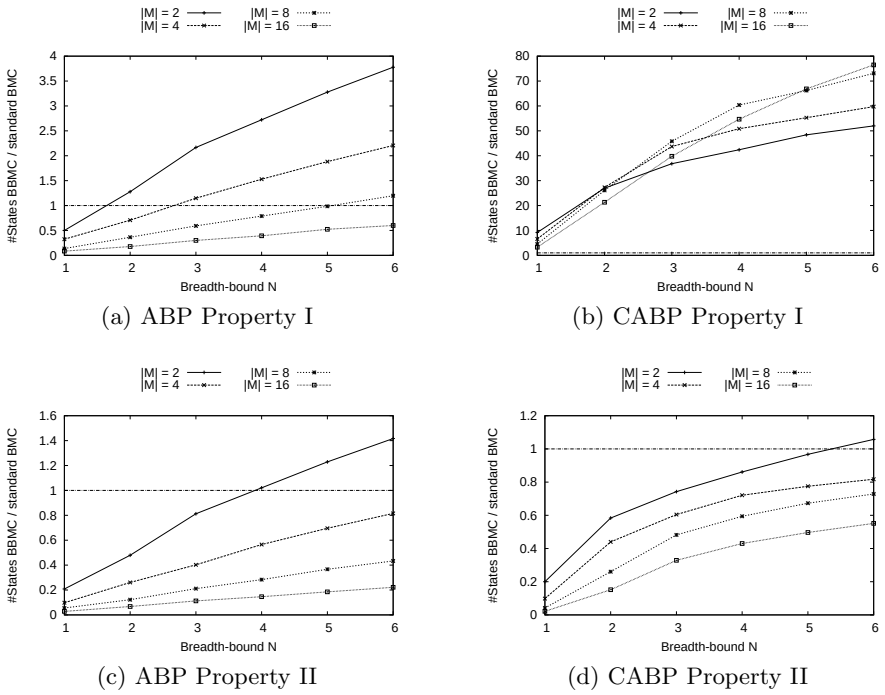
Analysis Combining Fig. 3 with the data from Table 1, we find that for property I, BMC is generally more effective than BBMC, and, even dramatically so for CABP. For property II, we find that for the ABP, BMC and BBMC are on a par for $|M| \leq 4$. For ABP with $|M| > 4$, and for all sizes of M for the CABP, BBMC clearly outperforms BMC.

The experiments confirm that BBMC and BMC are complementary, in the sense that (1) BBMC is in general less sensitive to an increase in state space size, compared to BMC, and, (2) BMC performs better on shallow properties, whereas BBMC is better suited for long-running and deep properties.

4.3 Experiment II

We next illustrate the effectiveness of our approach on a system of moderate size, a *cache coherence protocol* by Steve German [3]. The Cache Coherence Protocol (CCP) consists of a central controller and a fixed number of clients that communicate with the controller. Its main objective is to ensure shared or exclusive access to the cache, upon request. The protocol is known to fail the liveness requirement that every request for exclusiveness is inevitably granted: after a request for exclusiveness it is possible to find an infinite execution path in which the request is never granted. We use BMC and BBMC to find such a path. The results for CCP with 3 clients, using BBMC are depicted in Fig. 4. Again, we have normalised these results w.r.t. BMC: below 1.0, BBMC produces fewer states than BMC; above 1.0, BBMC produces more states than BMC. The probability of finding a witness is depicted in Fig. 4a.

Analysis Finding a witness using BMC requires at least 2,673 states, which is slightly more than 9% of the total state space. Fig. 4 clearly indicates that with little resources, BBMC is able to find a witness for the violation of the liveness requirement: at breadth bound $N = 1$, it requires roughly 70 states, *i.e.*, less than 0.25% of total state space, and it has a 60% chance of finding a witness in this case. At roughly 3% of the total state space, this probability is virtually 100%. Clearly, BBMC outperforms BMC in this example.



(a) ABP Property I (b) CABP Property I

(c) ABP Property II (d) CABP Property II

Fig. 3: Ratio of number of states produced by BBMC / BMC on the Alternating Bit Protocol and the Concurrent Alternating Bit Protocol with a set M of different messages. Ratio's below 1 indicate that BBMC produces fewer states than standard BMC requires for finding a witness to the property.

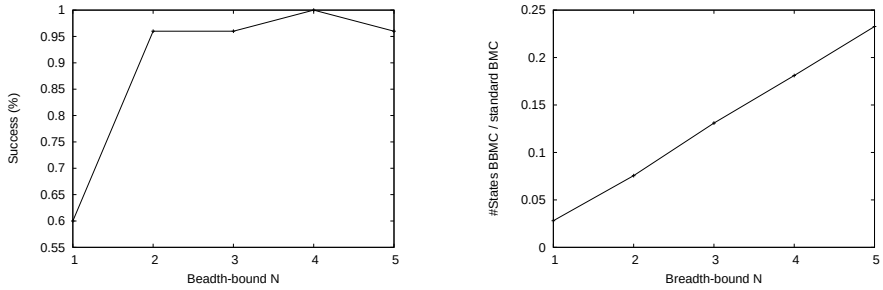
4.4 Experiment III

The next experiment illustrates the effects of applying *graceful degradation* in a verification. We experimented on an industrial *distributed lift system* with three lifts for lifting trucks. Two formal specifications of the distributed lift system are described in the literature, see [12]. The version we consider here formalises the system as it was originally implemented; it is known to fail five requirements. Using BMC and BBMC, we check for witnesses of these flaws for two out of these five requirements. In particular, we check for the following two requirements:

I a lift is not allowed to move up if the *up* button was not pressed prior to this event;

II pressing the *up* button, without releasing it, should eventually lead to all lifts moving upwards.

We illustrate the effect of *graceful degradation* on (1) the probability of finding a witness to these flaws and (2) the number of states constructed using BBMC. The results of our verification are depicted in Fig. 5.



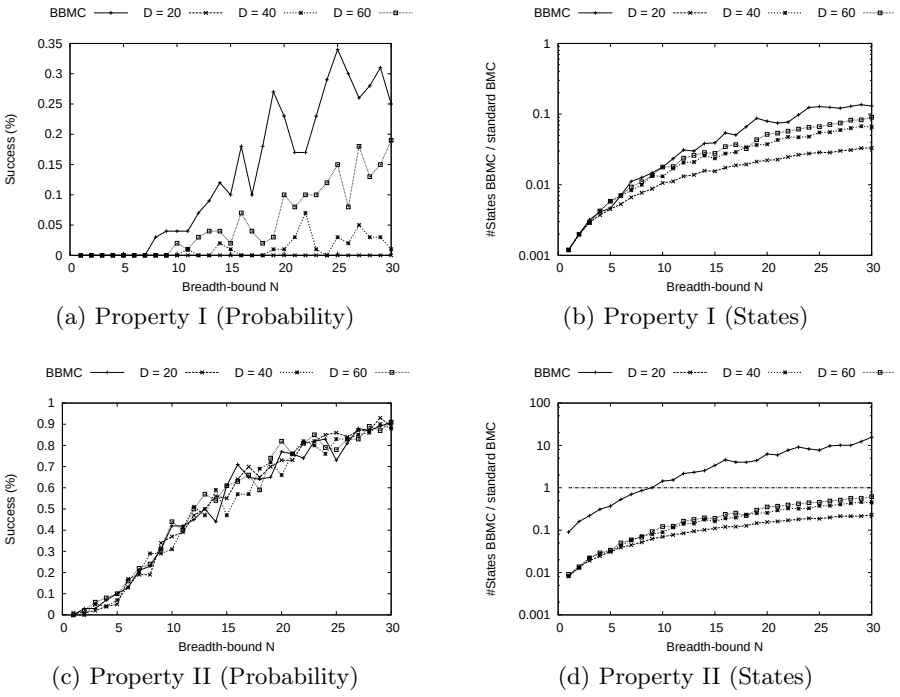
(a) Probability (b) States

Fig. 4: Measurements for the Cache Coherence Protocol. Depicted are the probability of finding a witness and the ratio of the number of states produced by BBMC / BMC at each breadth bound. BMC requires at least 2,673 states to be explored.

Analysis First, as we increase the graceful degradation bound from 20 to 60, the sizes of the resulting state spaces start approaching the sizes of the state spaces generated by Algorithm 2, *i.e.*, without graceful degradation (see Fig.s 5b and 5d; mind the logarithmic scale on the vertical axis). Moreover, we see that in practice, the average size of the state space generated by Algorithm 3 stays well below the estimated bound, which is obtained by multiplying the graceful degradation bound D, and the breadth-bound N. For instance, for $D = 20$ and $N = 10$, the average state space size is 80 states (well below the estimated bound of 200), and for $D = 60$ and $N = 25$, it is 501 states (again, well below the estimated bound of 1500).

Violations of properties that are more readily found at greater depths in the state space are harder to find by restricting the depth bound, see Fig. 5a. A graceful degradation bound of 20 even fails to find any. The results for the second property are more in favour of graceful degradation: compared to BBMC without graceful degradation, using graceful degradation has no adverse effects on the probability of finding a violation. This can be explained from the fact that a counterexample to property II can be found at shallower depths in the state space. We furthermore note the positive effect of a graceful degradation on the effectiveness for checking property II: in the worst case, BBMC with graceful degradation bound 60 and breadth-bound 30, 60% of the number of states of BMC requires, are generated, with a 90% effectiveness of finding a witness.

Lastly, we observe that without graceful degradation, BBMC outperforms BMC for property I, whereas for property II, the results are in favour of BMC. With graceful degradation enabled, this is reversed: BBMC with graceful degradation performs worse than BMC for property I, whereas for property II the results are more in line with BMC.



(a) Property I (Probability)

(b) Property I (States)

(c) Property II (Probability)

(d) Property II (States)

Fig. 5: Measurements for the original distributed lift system with 3 lifts. Depicted are the probability of finding a witness and the ratio of the number of states produced by BBMC / BMC at each breadth bound. BMC requires 7,536 states, resp. 1,101 states to find a witness for Property I, resp. Property II

5 Related Work

A main problem in verification is the state space explosion, which adversely affects the applicability of model checking. Over the years, several approaches for dealing with this issue have been investigated. Incomplete methods to model checking, such as our BBMC, are popular solutions. Among these methods, we find *Bounded Model Checking* (BMC) as pioneered by Biere *et al* [4, 6]. As explained in the introduction, BMC unfolds a symbolic transition relation up-to a pre-defined bound and encodes it as a propositional formula. The novelty of BMC lies in the fact that it relies on efficient SAT solvers to verify/refute a property. This contrasts our BBMC method, which is currently based on explicit state model checking. A main line of research on BMC is to find more efficient and effective SAT encodings of the problem by employing clever reduction techniques.

While early BMC approaches were predominantly based on the refutation of LTL formulae, follow-up works have also considered branching time logics such as ACTL [20] and even the universal fragment of the μ-calculus, *viz.* $\Box L_\mu$ [19]. The latter is in full agreement with our methods, which also allow one to refute $\Box L_\mu$ formulae (or, dually, confirm $\Diamond L_\mu$ formulae).

Contrary to BMC, our BBMC method relies on randomisations to achieve a greater effectiveness and a better coverage of the state space. Randomisation has been used prior to our work on BBMC. For instance, Grosu and Smolka, [14] have demonstrated that using random walks, it is possible to effectively find counterexamples to LTL properties in specifications. This has led to the development of *Monte Carlo Model Checking*, MC^2. Like BBMC, MC^2 relies on explicit state space model checking techniques. Furthermore, Grosu and Smolka present a technique to determine a bound on the number of attempts needed to achieve success. The latter technique seems unrelated to the use of random walks (instead of a probability space consisting of lasso's, one could replace it with the sub Kripke Structures that can be generated using BBMC); it is more related to the randomness in the approach *per se*. We thus believe that the same techniques are applicable to our work. Extending BBMC in this direction may be fruitful.

A critical note to the use of random walks is that these are known to give rise to a poor coverage of a state space, see *e.g.* [21, 10]. This is largely due to revisits of states; clearly, such revisits do not contribute to the coverage of the state space, and have adverse effects on the effectiveness of search techniques, see [10]. We conjecture that a similar observation can be made for model checking based on random walks, but experiments have yet to confirm this conviction.

The Highway search technique of [10] can be seen as a special case of BBMC, where properties are restricted to the form $\mu X.\Diamond X \vee p$ and p is an atomic proposition. There are some minor differences in the underlying mechanisms. For instance, we imposed an additional restriction on the original Highway simulation algorithm in order to guarantee totality of the generated state spaces, *i.e.*, our algorithms do not introduce sink states that were not present in the original specification. Highway search does not require one to do so. In fact, Highway search does not require one to construct a state space in the first place.

BBMC can be seen as a rudimentary form of *directed* model checking (DMC), see *e.g.* [11, 7, 9]. DMC guides a search using additional heuristics, and one hopes to direct the search effort to the bug-containing part of a state space without wasting time in searching the bug-free part of the state space. In this view, BBMC's heuristic is a blunt one: each state has equal probability of appearing in the resulting state space. The advantage is that, contrary to most DMC methods, our technique is universally applicable in the sense that it requires little human ingenuity, while it is still quite effective in finding corner-case bugs. The performance of DMC typically depends on the chosen heuristic.

Lastly, we briefly mention several other works for dealing with the state space explosion. Noteworthy are the approaches to effectively increasing memory sizes by partly flushing to high latency devices such as disks. Tailored algorithms have been studied that are I/O efficient, see *e.g.* [2], some of which are based on random explorations, see Abed *et al* [1]. Alternatively, the state space exploration can be distributed over several processing environments, either via grids or via multi-core processors. Exponents of such exploration methods are described in [24, 8, 16]. Some, methodologies are even based on combining several exploration techniques, see [23], and *SWARM*, see [15]. The latter basically or-

chestrates several different exploration techniques, each running on a dedicated processing unit. We believe that BBMC could be integrated efficiently in such an approach.

6 Conclusions

We described an alternative approach to *Bounded Model Checking*, in which we limited the breadth of exploration rather than the depth of exploration. The rationale behind our alternative approach is that, at least whenever software is involved, both *shallow* properties and *deep* properties must be analysed. While standard Bounded Model Checking is typically good at exploring shallow properties, deep properties require substantially more computational resources. This is due to the breadth-first exploration approach taken in Bounded Model Checking. Reaching greater depths is considerably more likely using our method of *Breadth-Bounded Model Checking* (BBMC).

Our BBMC method is based on randomised explicit state space exploration techniques, in which we rely on our algorithm to generate a state space that is a simulation of the full state space. This enables us to reuse the results of [17] which guarantee that any witness to an existential μ-calculus formula found in the simulation is also in the full state space. This is on a par with the capabilities of BMC [20]. We moreover showed that our algorithm preserves totality of the transition relation, *i.e.*, any sink state that is found in our simulation points at the reachability of a sink in the full state space.

Using a prototype implementation of the two algorithms that we provide in this paper, we conducted several experiments in which we verified systems of increasing size. The results of these experiments confirmed our hypothesis that the effectiveness of BBMC is most apparent for deep properties and "long-running" properties and BMC is generally better suited for shallow properties. This means the two techniques are complementary and should be exploited accordingly.

As for future issues for research, we are currently investigating how BBMC can be integrated efficiently in the model checking approach via Parameterised Boolean Equation Systems [13]. Moreover, we believe similar techniques can be applied to Parity Games, leading to alternative approaches for reducing the complexity of verification problems encoded as Parity Games.

Acknowledgements The authors would like to thank Michel Reniers and Hans Zantema for feedback on a preliminary version of this paper. This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.602 and ITEA2/TWINS.

References

1. N. Abed, S. Tripakis, and J.-M. Vincent. Resource-aware verification using randomized exploration of large state spaces. In *SPIN'08*, volume 5156 of *LNCS*. Springer, 2008.

2. J. Barnat, L. Brim, P. Simecek, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In *Proceedings of TACAS*, volume 4963 of *LNCS*, pages 48–62. Springer, 2008.
3. K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *Proceedings of VMCAI*, volume 2294 of *LNCS*, pages 317–330. Springer, 2002.
4. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
5. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT-Press, 1999.
6. E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
7. M. Torabi Dashti and A.J. Wijs. Pruning state spaces with extended beam search. In *Proc. ATVA'07*, volume 4762 of *LNCS*, pages 543–552. Springer, 2007.
8. M.B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proceedings of ICSE'07*, pages 3–12. IEEE Comp. Soc., 2007.
9. S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *Proceedings SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.
10. T.A.N. Engels, J.F. Groote, M.J. van Weerdenburg, and T.A.C. Willemse. Search algorithms for automated validation. *J. Log. Algebr. Program.*, 2009. In Press.
11. A. Groce and W. Visser. Heuristics for model checking java programs. *Software Tools for Technology Transfer*, 6(4):260–276, 2004.
12. J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *J. Log. Algebr. Program.*, 55(1-2):21–56, 2003.
13. J.F. Groote and T.A.C. Willemse. Parameterised Boolean Equation Systems. *Theor. Comp. Sc.*, 343:332–369, 2005.
14. R. Grosu and S.A. Smolka. Monte carlo model checking. In *Proceedings of TACAS*, volume 3440 of *LNCS*, pages 271–286. Springer, 2005.
15. G.J. Holzmann, R. Joshi, and A. Groce. Tackling large verification problems with the SWARM tool. In *SPIN'08*, volume 5156 of *LNCS*. Springer, 2008.
16. M.D. Jones and J. Sorber. Parallel search for LTL violations. *Software Tools for Technology Transfer*, 7:31–42, 2005.
17. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
18. S. Mauw and G. J. Veltink, editors. *Algebraic specification of communication protocols*. Cambridge University Press, 1993.
19. R. Oshman. Bounded model-checking for branching-time logic. Master's thesis, TECHNION - Israel Institute of Technology, 2008.
20. R. Oshman and O. Grumberg. A new approach to bounded model checking for branching time logics. In *ATVA*, volume 4762 of *LNCS*, pages 410–424. Springer, 2007.
21. R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *Proceedings FMICS'05*, pages 98–105. ACM Press, 2005.
22. W. Penczek, B. Wozna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundam. Inform.*, 51(1-2):135–156, 2002.
23. J.I. Rasmussen, G. Behrmann, and K.G. Larsen. Complexity in simplicity: Flexible agent-based state space exploration. In Orna Grumberg and Michael Huth, editors, *Proceedings of TACAS*, volume 4424 of *LNCS*, pages 231–245. Springer, 2007.

24. H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory checking. In *Proceedings of Parallel and Distributed Model Checking (PDMC'03)*, volume 89(1) of *ENTCS*, pages 51–67. Elsevier, 2003.