Bounded Model Checking

Henrik Torland Klev

November 2019

1 Introduction

My initial idea was to use chapter 10 "SAT-based Model Checking" of [1] as a basis for this presentation. However, said chapter turned out to be more of a reference-guide to more intriguing sources. After some pondering research, I came to the conclusion that, when available, the primary source is usually a good place to start. This lead me to the 1999 paper [2] where they propose using boolean decision procedures as an alternative to the traditional "Binary Decision Diagram"-based symbolic representation methods. However, some concepts, like wine and cheese, need time to mature into a better version of themselves. Even though the concepts of bounded model checking are perfectly outlined in the 1999 paper, a few years later a revised edition was published [3]. This newer versions gives a broader view of BMC in the grand scheme of verification, along with a section on completeness intended to satisfy critics of non-complete verification. As such, this presentation will focus primarily on the 2003 version, and is intended to provide a more context-based understanding of the field at that time. For a more detailed explanation, the reader is referred to the paper itself. Finally, it feels as if a disclaimer belongs here; as this is merely a student presentation, it should not be considered as part of the official curriculum unless explicitly approved by the courses lecturers.

2 Motivation

What common characteristics is shared by Michael Jackson's "Thriller", the dissolving of ATT's stable US monopoly, and the Falklands War? They are all far less interesting than the two main events of 1982, namely [4] and [5]. One interesting anecdote regarding Model Checking is that it was "invented" twice in the same year by two different duos. However, this is by far the most interesting feature of model checking. What makes it such an interesting subject is its ability to exhaustively examine the reachable stats of a program, its guaranteed termination for finite state-spaces, and its ability to produce counterexamples. As such, it can be produce a certificate of correctness (with regard to a specification), or, if faulty, give concrete, replicable proof of a bug; all while being immune to the halting problem [6]! Model checking employs algorithms that use instructions of a system under test to generate sets of states that are to be analyzed. These states must be stored to ensure that they are visited at most once. Unfortunately, the number of states exhibits exponential growth. The state-space can grow so prohibitively large that the challenge of storing them has been given its own name: the state-space explosion problem.

Attempts to combat this challenge has resulted in some notable breakthroughs, and it was the introduction of *symbolic model checking* that made the first step towards a wider acceptance in the industry. Symbolic model checking differs from the original explicit-state model checking techniques employs higher levels of abstraction in order to store less states, in contrast to storing each individual state in isolation. As an example of the difference between the two camps, consider the following recursive function:

```
function sum(x){
    if (x <= 0){
        return x;
    }
    sum(x-1);
}
sum(20);
```

Here, an explicit-state model checker would store the states

```
x = 20,
x = 19,
x = 18,
x = 17,
  .
  .
  .
x = 0
```

while the symbolic model checker would store the set of states internally as

```
0 <= x <= 20.
```

Within the camp of explicit-state model checking, an influential paper was released in 1996 introducing Partial Order Reduction as a way to combat the state-explosion problem [7]. However, as we focus on symbolic model checking, we are more interested in the events of 1986 [8] (BDDs), 1999[2] (BMC), and 2000[9] (CEGAR).

In addition to the two branches of model checking (explicit-state and symbolic), there is an independent verification method known as State Analysis built on the formal basis of Abstract Interpretation [10]. The key selling point of state analysis is its efficiency. [11] did a comparison between a state analysis tool known as "Parfait" with a bounded model checking tool "CBMC". Tests were performed on the BegBunch benchmarking framework provided by Sun Labs at Oracle. The results can be seen in figure 1. Notice the trade-off between efficiency and accuracy.

Iowa		
Bugs:421 Benchmarks:415		
	CBMC	Parfait
True Positive	413	274
False Positive	0	0
False Negative	8	147
True Negative	-	60
Running Time	18:40:59	00:00:47
Memory Peak	2.497 Gb	6712 Kb
Accuracy	98%	65%

(a) Iowa Verification Results

Samate		
Bugs:1033 Benchmarks:997		
	CBMC	Parfait
True Positive	998	866
False Positive	0	0
False Negative	35	167
True Negative	-	290
Running Time	01:16:18	00:01:26
Memory Peak	307.388Mb	6748 Kb
Accuracy	97%	84%

(b) Samate Verification Results

Cigital		
Bugs:11 Benchmarks:11		
	CBMC	Parfait
True Positive	11	0
False Negative	0	11
False Positive	0	0
True Negative	-	1
Running Time	00:01:50	00:00:01
Memory Peak	94.668 Mb	4372 Kb
Accuracy	100%	0%

(c) Cigtail Verification Results

Overall		
Bugs:1465 Benchmarks:1423		
	CBMC	Parfait
True Positive	1422	1140
False Negative	43	325
False Positive	0	0
True Negative	-	351
Running Time	19:59:07	00:02:14
Memory Peak	2.497 Gb	6748 Kb
Accuracy	97%	77%

(d) Overall Verification Results

Figure 1: Various Verification Results

Before we continue, it can be beneficial to take a step back to remember why we go through all this trouble. What is our goal? Our goal is to provide a rigorous guarantee of quality, in a highly automated and scalable way, to cope with the enormous complexity of software systems. The next section will introduce Bounded Model Checking as a way to reach our goal.

3 Bounded Model Checking

Why should one pick BMC as the formal technique for verification as opposed to more traditional BDD-based methods? This question can be a bit misleading; BMC does not provide the same certificate or guarantee of correctness as that of its unbounded counterpart. BMC sacrifices verification on behalf of finding (minimal) counterexamples. Therefore, if a strong guarantee of correctness is sought after, consider reading up on the traditional approaches instead. I suggest that the better question to ask is: when should one use BMC as a complement to BDD? One of the main drawbacks of BDDs is that it might not terminate; sometimes the state-space is too large, and might even be infinite. In such a scenario, the practical solution is to run BDD until you run out of time and/or space before you claim that you have verified the system. BMC, on the other hand, is more efficient at finding counterexamples due to its breadth-first approach and the abolished need for manual user intervention. In addition, the counterexamples are of minimal length, which makes them easier to understand (and fix). Therefore, if the system is too large for BDD-verification, or there is little faith in the system due to the (assumed large) presence of bugs, BMC will be a useful, efficient tool.

The idea behind bounded model checking is as follows:

Search for counterexamples in executions whose length is bounded by some integer k. If counterexample is found, return it. If not, increase k until problem becomes intractable, or you have reached the *Completeness Threshold*.

The BMC problem can be reduced to SAT (which have become really efficient), and a k between 60 and 80 outperformed BDD-based techniques in 2003.

Before we dive into the semantics of BMC, we need to refresh some concepts. To begin with, we start by defining the logic used for specifications - Linear Temporal Logic, LTL.

3.1 Definition of Linear Temporal Logic

Assume we have an infinite, countable set of Boolean propositions P. We define a model σ for a formula ϕ as an infinite sequence of truth of truth assignments to propositions. Given a model $\sigma = \sigma_0, \sigma_1, ...$, we denote the set of propositions at position i as σ_i. LTL formulas are constructed using a combination of the regular Boolean connectives and temporal operators:

$$\phi \ ::= \ p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid X\phi \mid G\phi \mid F\phi \, .$$

Here, X is the temporal operator *next*, G is *Global* and F is *Finally*. These are equivalent to the notations seen previously; they correspond to $\bigcirc$, $\Box$ and $\Diamond$ respectively. Further, for a formula ϕ and a position $i \geq 0$, ϕ hold at position i of σ, defined inductively as:

- For $p \in P$ we have $\sigma, i \models p$ iff $p \in \sigma_i$.
- $\sigma, i \models \neg\phi$ iff $\sigma, i \not\models \phi$.
- $\sigma, i \models \phi_1 \wedge \phi_2$ iff $\sigma, i \models \phi_1$ and $\sigma, i \models \phi_2$.
- $\sigma, i \models X\phi$ iff $\sigma, i+1 \models \phi$.
- $\sigma, i \models G\phi$ iff for all $i \geq 0$, $\sigma, i \models \phi$.
- $\sigma, i \models G\phi$ iff for some $i \geq 0$, $\sigma, i \models \phi$.

Note that the other Boolean operators (e.g., or, implication) and temporal operators (e.g., until, release) can be composed by a combination of the other introduced operators.

We wrap up this subsection by reminding the reader of the two main types of system properties one is interested in when regarding verification:

- Safety properties: what should always (not) happen, $G\phi$.
- Liveness properties: what should eventually happen, $F\phi$.

3.2 Definition of Kripke Structures

Kripke structures is one of the most popular formalism for representing transition systems in the context of model checking, and will be used extensively throughout the rest of this presentation.

Definition 3.1 (Kripke structures) *A Kripke Structure is a quadruple of the form $M = (S, I, T, L)$ where S is the set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and $L : S \to P$ is the labeling function, where P is the powerset of the atomic propositions. Labeling is a way to attach observations to the system: for a state $s \in S$ the set $L(s)$ is made of the atomic propositions that hold in s.*

3.3 Semantics for Bounded Model Checking

Before we can introduce the the semantics of BMC fully, we still have three more concepts we have to introduce: paths, witnesses and k-loops. Let us do them one by one:

Definition 3.2 (Paths) *Each path π in M is a sequence $\pi = (s_0, s_1, ...)$ of states, given in an order that respect the transition relation in M (i.e., T). If s_0 is an initial state, the path is initialized. The length of π, i.e. $|\pi|$, can be finite of infinite. For simplicity, we assume the set of initial states is non-empty, and that the transition relation T is total.*

Definition 3.3 (Witness) *LTL formulas are defined over all paths. Finding counterexamples corresponds to fining a contradicting trace. If we find such a trace, we call it a witness for the property. E.g., Gp corresponds to the question whether there exists a witness to $F\neg p$.*

Definition 3.4 (k-loops) *For $l \leq k$ we call a path π a (k, l)-loop if $T(\pi(k), \pi(l))$ and $\pi = u \cdot v^{\omega}$ with $u = (\pi(0), ..., \pi(l-1))$ and $v = (\pi(l), ..., \pi(k))$. We call π a k-loop if there exist a $k \geq l \geq 0$ for which π is a (k, l)-loop.*

Intuitively, we call a path π a (k, l)-loop if there is a transition from state k to state l and π is composed of the states 0 to $l-1$ followed by an infinite repetition of the states l to k. For illustrations, refer to the PowerPoint-presentation.

The semantics for a path *with* a loop are pretty boring, and needs little further explanation as long as you understand the semantics of LTL. They are explained in definition 3.5 and figure 2 below.

Definition 3.5 (Bounded Semantics for a Loop) *Let $k \geq 0$ and π be a k-loop. Then an LTL formula f is valid along the path π with bound k iff $\pi \models f$.*

$\pi \models p$	iff	$p \in L(\pi(0))$
$\pi \models \neg f$	iff	$\pi \not\models f$
$\pi \models f \wedge g$	iff	$\pi \models f$ and $\pi \models g$
$\pi \models \mathbf{X}f$	iff	$\pi_1 \models f$
$\pi \models \mathbf{G}f$	iff	$\pi_i \models f$ for all $i \geq 0$
$\pi \models \mathbf{F}f$	iff	$\pi_i \models f$ for some $i \geq 0$
$\pi \models f\mathbf{U}g$	iff	$\pi_i \models g$ for some $i \geq 0$ and $\pi_j \models f$ for all $0 \leq j < i$
$\pi \models f\mathbf{R}g$	iff	$\pi_i \models g$ if for all $j < i$, $\pi_j \not\models f$

Figure 2: Semantics for a path π containing a (k, l)-loop

However, if the path is not a k-loop, we have to be more pessimistic. Remember how we define validity: the formula $f := Fp$ is valid along π in the *unbounded* semantics if we can find an index $i \geq 0$ such that p is valid along the suffix π_i of π. However, in the *bounded* semantics, the state $\pi(k+1)$ does not exist. Therefore, we cannot define the bounded semantics recursively over suffixes of π. We therefore introduce a new notation $\pi \models_k^i f$, where i is the current position in the prefix of π, which means that the suffix π_i of π satisfies f. This leads us to definition 3.6 and figure 3.

Definition 3.6 (Bounded Semantics without a Loop) *Let $k \geq 0$ and π be a path that is not a k-loop. Then an LTL formula f is valid along π with bound k iff $\pi \models_k^0 f$.*

Before we delve into the next section, we provide evidence of how the existential model checking problem can be reduced to a bounded existential model checking problem. These lemmas (and the corresponding theorem) are proven elsewhere.

Lemma 3.1 *If a LTL formula is valid along a path with a bound, it is valid along a path without a bound.*

Lemma 3.2 *If a Kripke structure validates an unbounded existential LTL formula, then there exist a $k \geq 0$ such that the Kripke structure validates the bounded existential LTL formula.*

Theorem 3.1 *A Kripke structure validates an unbound existential LTL formula iff there exist a $k \geq 0$ such that the Kripke structure validates the bounded existential LTL formula.*

$$\begin{array}{lll}
\pi \models_k^i p & \textit{iff} & p \in L(\pi(i)) \\
\pi \models_k^i \neg p & \textit{iff} & p \notin L(\pi(i)) \\
\pi \models_k^i f \wedge g & \textit{iff} & \pi \models_k^i f \textit{ and } \pi \models_k^i g \\
\pi \models_k^i f \vee g & \textit{iff} & \pi \models_k^i f \textit{ or } \pi \models_k^i g \\
\pi \models_k^i \mathbf{G} f & \multicolumn{2}{l}{\textit{is always false}} \\
\pi \models_k^i \mathbf{F} f & \textit{iff} & \exists j,\ i \leq j \leq k.\ \pi \models_k^j f \\
\pi \models_k^i \mathbf{X} f & \textit{iff} & i < k \textit{ and } \pi \models_k^{i+1} f \\
\pi \models_k^i f \mathbf{U} g & \textit{iff} & \exists j,\ i \leq j \leq k.\ \pi \models_k^j g \textit{ and } \forall n, i \leq n < j.\ \pi \models_k^n f \\
\pi \models_k^i f \mathbf{R} g & \textit{iff} & \exists j,\ i \leq j \leq k.\ \pi \models_k^j f \textit{ and } \forall n, i \leq n < j.\ \pi \models_k^n g
\end{array}$$

Figure 3: Semantics for a path π which does not contain a (k, l)-loop

4 Transforming BMC to SAT

Although we are done with defining BMC, one major part has been left out until now. Let us now talk about how to reduce the bounded model checking problem into a Boolean satisfiability problem in order to take advantage of the state-of-the-art efficiency provided by modern SAT-solvers. Our new sub-goal as follows: Given a Kripke structure M, an LTL formula f and a bound k, construct a propositional formula $[M, f]_k$ that is satisfiable iff π is a witness for f. In short, the goal is accomplished by equation 1.

$$[M, f]_k := [M]_k \wedge \left(\left(\neg L_k \wedge [f]_k^0\right) \vee \bigvee_{l=0}^{k} \left({}_lK_k \wedge_l [f]_k^0\right) \right) \tag{1}$$

The first part (left of conjunction) constraints the path to be valid with regards to the transition relation in M starting from an initial state. The second part of the equation (left of disjunction) is the translation for a path without a loop, while the third and final part (right of disjunction) is the translation for a path with a loop. These parts will now be discussed in isolation.

Definition 4.1 (Unfolding of the Transition Relation) *For a Kripke structure M, and a given $k \geq 0$:*

$$[M]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

This definition is rather simple to understand intuitively; the reader is referred to the PowerPoint for an illustration.

Before we can define the translation with a loop, with need two to define some concepts in advance. Notice how we choose to define the third part of equation 1 before the second part. This is a consequence of the relation between the parts; the second part is a simplified version of the third.

Definition 4.2 (Loop condition) *The loop condition L_k is true iff there exist a back loop from state s_k to a previous state or to itself:*

$$L_k := \bigvee_{l=0}^{k} {}_lL_k$$

Definition 4.3 (Successor in a loop) *Let k, l, i be non-negative integers such that $l, i \leq k$. The successor $succ(i)$ of i in a (k, l)-loop is defined as*

$$succ(i) := i + 1 \mid i < k$$
$$succ(i) := l \mid i = k$$

Now we have all the parts we need in order to introduce the translations of LTL formulas into propositional formulas.

Definition 4.4 (Translation of an LTL formula with a loop) *Let f be an LTL formula, and $k, l, i \geq 0$, qith $l, i \leq k$. The semantics are then seen in figure 5.*

$$\begin{array}{llll}
{}_l[\![\, p \,]\!]_k^i & := p(s_i) & {}_l[\![\, \mathbf{G}f \,]\!]_k^i & := {}_l[\![\, f \,]\!]_k^i \wedge {}_l[\![\, \mathbf{G}f \,]\!]_k^{\mathrm{succ}(i)} \\
{}_l[\![\, \neg p \,]\!]_k^i & := \neg p(s_i) & {}_l[\![\, \mathbf{F}f \,]\!]_k^i & := {}_l[\![\, f \,]\!]_k^i \vee {}_l[\![\, \mathbf{F}f \,]\!]_k^{\mathrm{succ}(i)} \\
{}_l[\![\, f \vee g \,]\!]_k^i & := {}_l[\![\, f \,]\!]_k^i \vee {}_l[\![\, g \,]\!]_k^i & {}_l[\![\, f\mathbf{U}g \,]\!]_k^i & := {}_l[\![\, g \,]\!]_k^i \vee ({}_l[\![\, f \,]\!]_k^i \wedge {}_l[\![\, f\mathbf{U}g \,]\!]_k^{\mathrm{succ}(i)}) \\
{}_l[\![\, f \wedge g \,]\!]_k^i & := {}_l[\![\, f \,]\!]_k^i \wedge {}_l[\![\, g \,]\!]_k^i & {}_l[\![\, f\mathbf{R}g \,]\!]_k^i & := {}_l[\![\, g \,]\!]_k^i \wedge ({}_l[\![\, f \,]\!]_k^i \vee {}_l[\![\, f\mathbf{R}g \,]\!]_k^{\mathrm{succ}(i)}) \\
 & & {}_l[\![\, \mathbf{X}f \,]\!]_k^i & := {}_l[\![\, f \,]\!]_k^{\mathrm{succ}(i)}
\end{array}$$

Figure 4: Translation of and LTL formula with a loop

Definition 4.5 (Translation of an LTL formula without a loop) *Same principles as definition 4.4 except succ(i) is simplified to i+1, and index l is no longer needed. The semantics are seen in figure* **??**.

Inductive Case: $\forall i \leq k$

$$\begin{aligned}
[\![p]\!]_k^i &:= p(s_i) & [\![\mathbf{G}f]\!]_k^i &:= [\![f]\!]_k^i \wedge [\![\mathbf{G}f]\!]_k^{i+1} \\
[\![\neg p]\!]_k^i &:= \neg p(s_i) & [\![\mathbf{F}f]\!]_k^i &:= [\![f]\!]_k^i \vee [\![\mathbf{F}f]\!]_k^{i+1} \\
[\![f \vee g]\!]_k^i &:= [\![f]\!]_k^i \vee [\![g]\!]_k^i & [\![f\mathbf{U}g]\!]_k^i &:= [\![g]\!]_k^i \vee ([\![f]\!]_k^i \wedge [\![f\mathbf{U}g]\!]_k^{i+1}) \\
[\![f \wedge g]\!]_k^i &:= [\![f]\!]_k^i \wedge [\![g]\!]_k^i & [\![f\mathbf{R}g]\!]_k^i &:= [\![g]\!]_k^i \wedge ([\![f]\!]_k^i \vee [\![f\mathbf{R}g]\!]_k^{i+1}) \\
& & [\![\mathbf{X}f]\!]_k^i &:= [\![f]\!]_k^{i+1}
\end{aligned}$$

Base Case:

$$[\![f]\!]_k^{k+1} := 0$$

Figure 5: Translation of and LTL formula without a loop

5 Completeness

Just in case you come across someone claiming that ”Bounded model checking is useless because it gives up completeness! You cannot use it to verify correctness!” this section provides ways of reclaiming completeness. Note that this part is more theoretical than practical, and if completeness is what you are after, BDD might be more down your alley.

5.1 The Completeness Threshold

For every finite state system M, a property p, and a given translation scheme, there exist a number CT, such that the absence of errors up to cycle CT proves that $M \models p$. For formulas of the form Gp, this is simply the number of steps required to reach all states. This is called the reachability diameter, and can be seen in figure 6. This is really just a formal way of stating that $rd(M)$ is the longest, ”shortest path” from an initial state to *any* reachable state.

$$\begin{aligned}
&rd(M) := \min\{i \mid \forall s_0, \ldots, s_n.\ \exists s'_0, \ldots, s'_t, t \leq i. \\
&I(s_0) \wedge \textstyle\bigwedge_{j=0}^{n-1} T(s_j, s_{j+1}) \rightarrow (I(s'_0) \wedge \textstyle\bigwedge_{j=0}^{t-1} T(s'_j, s'_{j+1}) \wedge s'_t = s_n)\}
\end{aligned}$$

Figure 6: Equation for calculating the reachability diameter

The equation in figure 6 is hard to solve for realistic models. However, it is possible to compute an over-approximation with a SAT instance (figure 7 which calculates the longest loop-free path in M starting from an initial state.

$$rdr(M) := max\{i| \exists s_0 \dots s_i .\ I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\}$$

Figure 7: Equation for calculating an over-approximation of the reachability diameter

5.2 Translation of Liveness Properties

So far we have focused on existentially quantified temporal logic formulas: to verify an existential LTL formula against a Kripke structure, one needs to find a witness for said property. In the case of liveness, the dual is also true: if a proof of liveness exist, it can be established by examining all finite sequences of length k starting from initial states. Formally, this is defined as follows:

Definition 5.1 (Translation of Liveness Properties)

$$[M, \forall Fp]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \rightarrow \bigvee_{i=0}^{k} p(s_i)$$

5.3 Induction

The last way of regaining completeness consists of proving safety properties by finding (manually) a strengthening *inductive invariant* - an invariant that is inductive (i.e., its current correctness relies on its correctness in previous steps), and implies the safety property in question. This is done over three steps (see figure 8, 9 and 10).

$$rdr(M) := max\{i| \exists s_0 \dots s_i .\ I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\}$$

Figure 8: 1. Check that the base-case is unsatisfiable.

$$\exists s_0, \dots, s_{n+1} .\ \bigwedge_{i=0}^{n} (\phi(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg\phi(s_{n+1})$$

Figure 9: 2. Check induction step is unsatisfiable.

$$\forall s_i .\ \phi(s_i) \rightarrow p(s_i)$$

Figure 10: 3. Establish that the strengthening inductive invariant implies the property for an arbitrary i.

6 Summary

During this presentation we have stated the need for symbolic model checking, defined the semantics for Bounded Model Checking, translated the BMC-problem to a SAT-problem, and finally we discussed how to regain completeness.

For any further questions or comments, feel free to contact me at henriktk@ifi.uio.no.

References

[1] *Handbook of Model Checking*. eng. Cham, 2018.

[2] A Biere et al. “Symbolic Model Checking without BDDs”. English. In: *Tools And Algorithms For The Construction And Analysis Of Systems* 1579 (1999), pp. 193–207. ISSN: 0302-9743.

[3] Armin Biere et al. “Bounded Model Checking”. In: *Advances in Computers* 58 (2003). URL: `http://fmv.jku.at/papers/BiereCimattiClarkeStrichmanZhu-Advances-58-2003-preprint.pdf`.

[4] J.P. Queille and J. Sifakis. “Specification and verification of concurrent systems in CESAR”. In: vol. 137. Springer Verlag, 1982, pp. 337–351. ISBN: 9783540114949.

[5] E.M. Clarke and E.A. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: vol. 131. Springer Verlag, 1982, pp. 52–71. ISBN: 9783540112129.

[6] Alan M. (Alan Mathison) Turing. “On computable numbers, with an application to the Entscheidungsproblem”. eng. In: New York, 1965, pp. 115–153.

[7] *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem*. eng. Berlin, Heidelberg, 1996.

[8] Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. eng. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. ISSN: 0018-9340.

[9] E. Clarke et al. “Counterexample-guided abstraction refinement”. In: vol. 1855. Springer Verlag, 2000, pp. 154–169. ISBN: 3540677704.

[10] P. Cousot and R. Cousot. “Abstract interpretation: ”A” unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: vol. 130756. Association for Computing Machinery, 1977, pp. 238–252.

[11] Kostyantyn Vorobyov and Padmanabhan Krishnan. “Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches”. In: Jan. 2010.