Saarland University

Faculty of Natural Sciences and Technology I
Department of Computer Science

Approximate LTL Model Counting

Bachelor's Thesis

Jennifer Maria Niederländer

Reviewers

Prof. Bernd Finkbeiner
Dr. Swen Jacobs

submitted
27.04.2015

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum / Date) (Unterschrift / Signature)

Abstract

We consider the model counting problem of linear-time temporal logic (LTL). LTL is a well known specification logic and a standard input language for model checking and synthesis tools of reactive systems. The LTL model counting problem distinguishes two types of models: word and tree models. Word models are labeled sequences that satisfy the formula and are bounded in length. Counting word models can be used in verification to compute the number of errors in an implementation. Tree models are labeled trees that are bounded by depth where each path satisfies the formula. Counting tree models can be seen as a quantitative extension of synthesis in which we determine the number of implementations that satisfy the formula. Unfortunately, the best counting algorithms we know has double exponential complexity in the formula for word models and threefold exponential complexity in the formula for tree models. Although this algorithm improves the naive solution in terms of complexity in the bound, both solutions are impractical and are mostly limited to formulas of small length. In this thesis, we investigate approximative algorithms that efficiently solve the LTL counting problem. We present two main techniques, namely a technique based on the ideas of bounded model checking, where both the model and the LTL formula are transformed into a SAT formula, and another technique based on Monte Carlo methods, where we compute a set of possible solutions and determine proper lower and upper bounds.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Model counting is the problem of counting the number of models that fulfil a certain logical formula. Model Counting has been mainly studied for propositional logic (SAT Counting) and contributed to several planning and probabilistic reasoning tasks, as computing the robustness of a plan [12] and Bayesian net reasoning [10]. Moreover, for SAT counting, research has already developed a large number of exact and approximate SAT counters like Relsat [1] and ApproxCount [18], as well as ApproxMC [4].

In this thesis, we study the model counting problem for formulas of linear-time temporal logic (LTL) [5]. LTL is one of the most used specification logics, especially for reactive systems [14]. In verification, model counting is used to determine the number of errors in a system and thus assesses the reliability of a system. Model counting can also be seen as a generalization of synthesis: instead of constructing a program that satisfies the specification, it determines the number of implementations that satisfy a specification. This number can be used to determine the precision of a specification by determining how much implementation freedom is left by the specification.

When counting the models of an LTL formula, we distinguish between two kinds of models: word and tree models. A word model of a LTL formula φ over a set of atomic propositions AP is a sequence from 2^{AP} such that this sequence satisfies φ. A tree model of φ over a set of atomic propositions, partitioned into in- and outputs, is a tree that branches according to the input values and is labeled with output values, such that every path of the tree satisfies the formula. Following the ideas of bounded model checking and bounded synthesis [2] the authors in [5] define the model counting problem over bounded models.
Algorithms for computing the number of models of an LTL formula have been introduced in [5]. As far as we know, this is the best LTL model counting approach so far. Unfortunately, it introduces high exponential complexity in the length of the formula, which makes it impractical, especially for large formulas or large bounds of the models. In this thesis, we therefore develop two new approaches for LTL model counting. The first approach is an approximate LTL counting approach based on Monte Carlo, the second approach is based on an encoding in propositional logic following the ideas of bounded model checking.

Counting algorithms based on Monte Carlo methods have already been developed for of propositional formulas [9, 15]: many of these algorithms can produce good estimation results, whereas the exact computation is very hard or impossible. One example for such algorithms is the DNF counting algorithm in [9]. Motivated by the good results for SAT counting, we aim at using

the Monte Carlo method for LTL counting by creating a certain amount of bounded models at random. For checking whether the word models fulfill the formula, we use the CTL model checking algorithm [11]. For tree models, we use the standard automata based model checking algorithm.

The second approach we introduce is based on the SAT encoding introduced in bounded model checking. Bounded model checking is a technique, that builds on the assumption that some hard problems for LTL checking are not that hard for SAT solving. It presents good results in solving instances that cannot be handled by competing techniques. Modern SAT solvers can deliver very good results and can handle thousands of variables. Inspired by these results, we aim to adapt the bounded model checking technique by transforming our models and our specifications into formulas of propositional logic. Afterwards, we feed the transformed SAT formula into one exact and two approximate SAT counters, respectively. The exact SAT counter we use is Relsat [1] and our approximate SAT counters are ApproxCount [18] and ApproxMC [4]. Since bounded model checking relies on an exponential procedure, it is limited in its capacity. Furthermore, since bounded model checking is bounded by some number k, this method may deliver incomplete results, if this bound is not large enough.

The rest of the thesis is structured as follows. In Chapter 2, we introduce the necessary knowledge about linear-time temporal logic, its models and the automata used in the presented approaches. Afterwards, in Chapter 3, we take a closer look at the model counting problem itself. Furthermore, in Chapter 4, we introduce our exact bounded model counter that uses Relsat [1] as well as the related approaches presented by Finkbeiner et al. [5]. In Chapter 5, we present our approximate approaches based on Monte Carlo and bounded model counting as well as the related work of Vardi et al. [4] and Wei et al. [18]. Next, we explain the implementations of our approaches and the exact LTL counting approach [5] in the following chapter. In Chapter 6, we compare the results of all implemented approaches.

CHAPTER 2

BACKGROUND

In this section we introduce the background knowledge needed to understand the following parts of the thesis. We take a look at linear-time temporal logic, its syntax, its semantics and the automata needed for our approaches.

2.1 Linear-time Temporal Logic

In order to express our specifications properly, we use Linear-time Temporal Logic (LTL). Temporal logic extends propositional logic by modalities that enable depicting the infinite behaviour of reactive systems. Its models are transition systems, which are defined as follows:

Definition 1 (Transition Systems) *A model of an LTL formula is a labeled transition system. Given a finite set of directions Υ and a finite set of labels Σ, a Σ-labeled Υ-transition system is a tuple $\boldsymbol{S} = (S, s_0, \tau, o)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $\tau : S \times \Upsilon \to S$ is the transition function and $o : S \to \Sigma$ is a labeling function. A path in a labeled transition system is expressed by a sequence $\pi : \mathbb{N} \to S \times \Upsilon$ of states and directions. This sequence follows the transition relation:*
For all $i \in \mathbb{N}$ if $\pi(i) = (t_i, e_i)$ then $\pi(i+1) = (t_{i+1}, e_{i+1})$, where $t_{i+1} \in \tau(t_i, e_i)$.
An initial path is a path that starts with the initial state $\pi(0) = (t_0, e)$ from some $e \in \Upsilon$. The set of paths($\boldsymbol{S}$) is defined as the set of all initial paths of $\boldsymbol{S}$.

The definition of LTL is described next:

Definition 2 (Syntax of Linear-time Temporal Logic) *We use LTL with the operators Next X, Until U, Release R, Eventually $\Diamond$ and Globally $\Box$. LTL formulas are defined over a set of atomic propositions $AP = I \cup O$. I denotes the set of input variables and O the set of output variables. A LTL formula φ over the set AP is formed according to the following grammar:*

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 U \varphi_2 \mid \varphi_1 R \varphi_2 \mid X\varphi \mid \Diamond\varphi \mid \Box\varphi$$

where $a \in AP$.

Now that we know the syntax of LTL formulas, we need to specify their semantics.

Definition 3 (Satisfiability of a Specification) *Satisfiability of a specification, expressed as an LTL formula φ, is denoted by a sequence $\sigma : \mathbb{N} \to 2^{AP}$ of valuations of the atomic propositions by $\sigma \models \varphi$. We say that a 2^O-labeled 2^I-transition system $\boldsymbol{S} = (S, s_0, \tau, o)$ satisfies the formula*

φ, if for all $\pi \in paths(\boldsymbol{S})$ the sequence $\sigma_\pi : i \to o(\pi(i))$, where $o(s,e) = (o(s) \cup e)$, satisfies φ. Figure 2.1 shows when a path fulfills a formula.

$$
\begin{array}{lll}
\sigma \models true & & \\
\sigma \models a & \text{iff} & a \in A_0 (i.e. A_0 \models a) \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\
\sigma \models X\varphi & \text{iff} & \sigma[1..] = A_1 A_2 A3... \models \varphi \\
\sigma \models \varphi_1 U \varphi_2 & \text{iff} & \exists j \geq 0. \sigma[j...] \models \varphi_2 \text{ and } \sigma[i...] \models \varphi_1, \text{ for all } 0 \leq i < j
\end{array}
$$

Figure 2.1: LTL semantics for infinite words over 2^{AP} [14]

Given any LTL formula, we can transform LTL into Positive Normal Form (PNF). In this form, the formula does not contain any Eventually $\Diamond$ or Globally $\Box$ operators. There are two forms of the PNF: The release PNF and the weak-until PNF. In our approach, we use the release PNF defined as follows:

Definition 4 (Release Positive Normal Form) *For $a \in AP$, a LTL formula in release positive normal form is given by*

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 U \varphi_2 \mid \varphi_1 R \varphi_2 \mid X\varphi$$

.
Formulas that contain Globally or Eventually operators are expressed as:

$$\Box\varphi \equiv falseR\varphi$$

and

$$\Diamond\varphi \equiv trueU\varphi$$

.

2.2 Automaton

In our approach we use automata to check the satisfiability of our LTL formulas. One such type is the class of generalized non-deterministic Büchi automata that is explained below.
Before actually defining general non-deterministic Büchi automata, we take a look at nondeterministic Büchi Automata [3] in Definition 5 and afterwards define generalized nondeterministic Büchi automata in Definition 6

Definition 5 (Nondeterministic Büchi Automata (NBA)) *A nondeterministic Büchi Automata (NBA) $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states, called the acceptance set. A run for $\sigma = A_0 A_1 A_2 ... \in \Sigma^\omega$ denotes an infinite sequence $q_0 q_1 q_2 ...$ of states in $\mathcal{A}$ such that $q_0 \in Q_0$ and $q_i \xrightarrow{A_i} q_{i+1}$ for $i \geq 0$. A run $q_0 q_1 q_2 ...$ is accepting, if $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$.*

The main difference between the NBA presented above and the GNBA presented below, is that even though both are equally expressive, the GNBA have a more general acceptance condition than "visit infinetly often the acceptance set F". In our case, the GNBA requires us to visit several acceptance sets $F_1 ... F_k$ infinitely often.

Definition 6 (Generalized nondeterministic Büchi Automata (GNBA)) *A generalized NBA (GNBA) is a tuple $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $\mathcal{F}$ is a subset of 2^Q, that can be empty.*
We call the elements $F \in \mathcal{F}$ acceptance sets. A run for $\sigma = A_0 A_1 A_2 ... \in \Sigma^\omega$ denotes an infinite sequence $q_0 q_1 q_2 ...$ of states in $\mathcal{A}$ such that $q_0 \in Q_0$ and $q_i \xrightarrow{A_i} q_{i+1}$ for $i \geq 0$. We call an infinite run $q_0 q_1 q_2 ...$ accepting if

$$\forall F \in \mathcal{F}.(\exists^\infty j \in \mathbb{N}.q_j \in F)$$

CHAPTER 3

THE MODEL COUNTING PROBLEM

The Model Counting problem is the problem of counting models that fulfill a certain specification. More specifically, in this thesis, we are regarding LTL counting, where the models are finite transition systems and the specification language is LTL. Another version of model counting is SAT counting, where the specification language is propositional logic and we count the number of truth assignments. Checking whether a model satisfies a specification is called model checking.
Note that model counting is much harder than model checking or satisfiability checking. Exact model counters can tackle formulas with about 100 variables, approximate model counters can tackle formulas with about 1000 variables [8]. Model counting can be very important in practice, because it can have a significant impact on many application areas such as planning, probabilistic reasoning and combinatorial problems. Moreover, it is a useful generalization of satisfiability.
In the following sections, we introduce both model counting problems in more detail, starting with SAT counting.

3.1 SAT counting

SAT counting introduces a new scalability challenge that receives more and more interest, while SAT solving has become one of the best automated reasoning techniques. Most SAT counting approaches are either based on DPLL, like Relsat [1], or build on local search. This led to a bunch of new ideas starting with the extension of DPLL-based methods to search sampling. However, there are also problems that cannot be tackled by SAT counting, because scaling is impossible. In general, these kinds of problems are also hard for SAT solving. The main challenge is making the approaches scale even for hard problems.

For SAT counting, there are several approaches that count the number of truth assignments. In Chapter 4 and 5, we introduce the SAT counters Relsat [1], ApproxCount [18] and ApproxMC [4]. In this section, we take a closer look at another SAT counting based on the Monte Carlo method. Rubinstein et al. [15] consider the counting problem for satisfiability checking by viewing counting problems as instances of estimation problems. Their main contribution is listing several counting algorithms that use the Monte Carlo method and introducing their own approach based on sampling and the MinxEnt method, as well as an IS algorithm for the SAT problem in DNF form. Moreover, they try to avoid the hardness of the counting problem by splitting up difficult counting problems into several easy counting problems.

Amongst others, they introduced Monte Carlo methods for the self-avoiding walk and several sampling problems and defined random K-SAT problems. Furthermore, they presented a counting algorithm for SAT formulas in DNF that has a low complexity. Finally, they explain the MinxEnt method that is non-parametric in contrast to classic CE methods, and introduce a parametric version of MinEnt, called PME. For evaluation, they consider some K-SAT problems and computed some comparative simulation results for both the standard CE method and PME, though PME clearly performs better. It is more accurate and for K-SAT problems with $K \leq 5$ it is clearly superior to the CE method. In addition, its relative error rate of 0.02% is much smaller than the relative error rate of the CE method being 0.6%.

Although this paper introduces some interesting new opportunities for SAT counting, it does not tackle our problem for LTL counting. Of course, it is possible to reduce LTL to propositional logic with help of the bounded model checking formulas and then we would be able to apply some of the methods presented here. However, since bounded model checking reduces our LTL formula and bounds it by some bound k, it is not the best for verification of a system. Moreover, translating the formula to propositional logic affords extra time and has to be done once for every different bound.

3.2 LTL counting

LTL counting introduces new challenges for model counting and provides a generalization of model checking and synthesis.
By taking a closer look at LTL model counting, we see that naively counting transition systems that satisfy a LTL formula is not sufficient. By counting the number of transition systems that fulfill a certain specification, we either get zero or ∞ as a result. The reason for this result is that, if the specification is satisfiable by some periodic transition system, we can unroll its periodic part infinitely often and each unrolling results in a new transition system that satisfies the formula.
In order to avoid this effect, we consider bounded models. Thereby, we distinguish between two types, word and tree models.
We consider the model counting problems for word models and tree models. After defining word models, we take a closer look at the model counting approach for word models in Chapter 4, originally presented by Finkbeiner and Torfah [5] . A word model is a sequence bounded by some value k with a loop-back transition from the last position in the sequence to a former position $i \leq k - 1$.

Definition 7 (Word Model) *We denote the lasso sequence $\pi(o) \dots \pi(i-1)(\pi(i) \dots \pi(k))^\omega \in (2^O x 2^i)^\omega$ for some $i \in \{0, \dots, k\}$ as a k-word model of a LTL formula φ over $AP = I \cup O$. We call $\pi_\perp = \pi(0) \dots \pi(k) \in (2^O x 2^i)^{k+1}$ the base of the word model.*

In Figure 3.1 you can see a base of a word model and two word models built upon that base.

Now, we can redefine our counting problem:

Definition 8 (k-word counting problem) *For a LTL formula φ and a bound k, the k-word counting problem is to compute the number of k-word models of φ.*

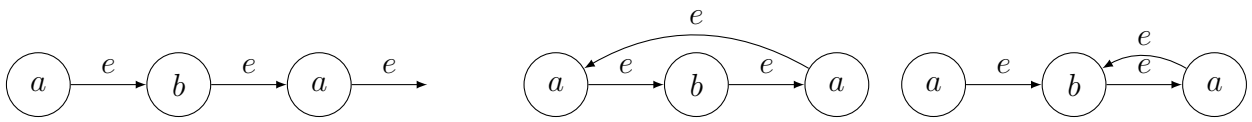


Figure 3.1: A base and two word models [5]

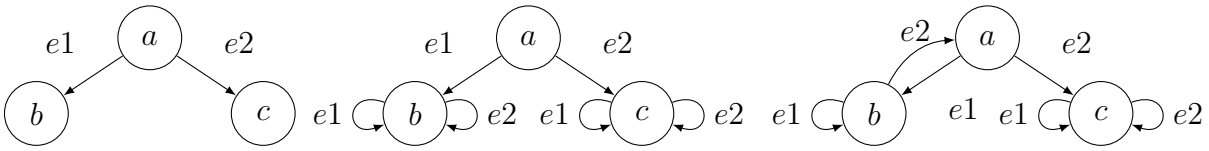


Figure 3.2: A base and two tree models [5]

Next, we define tree models and in Chapter 4, we recall the model counting approach for tree models originally presented by Finkbeiner and Torfah [5].
As a tree model we denote a tree that branches according to the valuations of the input I and that is labelled with the valuations of the output O such that every path satisfies the LTL formula φ over a set of atomic propositions $AP = I \bigcup O$.

Definition 9 (Tree Model) *We denote a 2^O-labeled-2^I-transition system that forms a tree of depth k with additional loop-back transitions from the leaves as a k-tree model of a LTL formula φ. Thereby, for every leaf and every direction, there is an edge to some state of the branch leading to the leaf. We call the tree without the loop-back transitions the base of the tree model.*

In Figure 3.2 you can see a base of a tree model and two tree models built upon that base.

With the Definitions 7 and 9 we can redefine our model counting problem:

Definition 10 (k-tree counting problem) *For a LTL formula φ and a bound k, the k-word/k-tree counting problem is to compute the number of k-word/k-tree models of φ.*

CHAPTER 4

EXACT MODEL COUNTING

In this chapter, we present approaches that aim at solving the model counting problem in an exact manner.

4.1 Counting Models for Linear Time Temporal Logic

First, we take a look at the two automata based approaches presented in [5]. We start with the explanation of their approach for word models and continue with explaining their approach for tree models.

4.1.1 Counting Word Models

The k-word model counting problem for LTL was already tackled in the work of Finkbeiner and Torfah [5], who also consider bounded models and developed an exact counting algorithm with linear complexity in the bound. It is obtained by dynamic programming. They inductively compute the number of models, to compute the number of models of the next size. Their approach is based on a translation to unambiguous word automata where each model has exactly one run in the automaton.

Unfortunately, their construction is expensive in the size of the formula. Therefore, their approach only scales for small specifications.

In the following paragraph, we take a closer look on their approach for word models.

Finkbeiner and Torfah count the word models for a bound k and a safety specification φ by constructing a word automaton that accepts a finite sequence of length k, if this sequence represents a base for a word model of the formula φ. They show that, for each LTL formula φ that represents a safety property and a bound k, it is possible to construct a word automaton that accepts a word of maximum length k, if it is a base of a word model. In order to construct such an automaton, we start with the representation of φ as an universal safety automaton.
In general, the universal safety automaton yields a run graph on a word model if it satisfies the specification φ. In a run graph, every state of the word model is mapped to a corresponding state in the universal safety automaton. We call the set of universal states visited by the word model in the state s the annotation of s. From the Definition 7 of word models, we know that a loop in a word model corresponds to a suffix of the base. Therefore, the annotation of this suffix corresponds to a repeating annotation in the run graph of the word model. The word automaton

starts with guessing the annotation of the loop-back state. By traversing the base backwards, it checks whether a repetition of the guessed annotation is observed and an initial annotation is reached after traversing the whole base. Moreover, the word automaton considers that one base may correspond to several word models and therefore keeps track of the number of repetitions of the guessed annotation. Furthermore, the automaton needs to be unambiguous with respect to a word model. This means that, although every base has at most one single annotation for a word model, a single base may have multiple annotations. In order to prevent this, the word automaton only allows maximal annotations.
Now, we take a closer look at the actual counting algorithm, presented in Algorithm 1. Basically,

Algorithm 1 Counting Word Models with $\mathcal{A}_{\#}$ [5]

$\Omega = \{(q, 1) \mid q \in Q_{0\#}\}$;
for $(i := 0, i \leq k, i{+}{+})$ **do**
  for all $(q \in \Omega)$ **do**
    for all $\sigma \in \Sigma \times \Upsilon$ **do**
      $\Omega(\Delta(q, \sigma)) + := \Omega(q)$
    end for
  end for
end for
return $q = \sum_{q \in Q_{F\#}} \Omega(q) * c$ (*$Q_{F\#}$ is the set of accepting states*)

the algorithm computes the number of models by counting the number of base annotations σ of length i for each state q that are accepted by q in the i-th iteration. Ω maps each accepting state in the k-th iteration to the number of bases of length k that are accepted by the automaton $\mathcal{A}_{\#}$. Note that each base has c word models. Finally the number of word models is computed by summing up the number of word models in each accepting state.

The algorithm traverses the automaton k times and thus, the complexity is $\mathcal{O}(k).2^{2^{\mathcal{O}(|\varphi|)}}$. This double exponential complexity is responsible for the limitation of this approach to small formulas. Since there are a lot of formulas with hundreds of variables used in practice, this approach cannot be used. We therefore develop an approximate algorithm with lower complexity. This approach is presented in Chapter 5.

4.1.2 Counting Tree Models

They also investigated the k-tree model counting problem. Again, they consider bounded models. A tree model is bounded by its depth. Their algorithm is based on a automata translation, where a universal safety automaton is translated into a bottom-up tree automaton. Like their algorithm for word models, the complexity of their algorithm is again linear in the bound. Unfortunately, the complexity in the length of the formula is threefold exponential. In the following paragraph, we take a closer look on their approach for tree models.

Similarly to their approach for word models, they start with the representation of the LTL formula φ as a universal safety automaton. They guess a loop annotation and check whether this annotation is repeated after exploring the tree from the leaves upwards. It is required to guess an annotation for each branch in the tree because a tree model is basically a composition of

word models.

By traversing the tree upwards, they apply the procedure for word models combined with an additional merging procedure that merges all information received from the children into their parent state. Thereby, a bottom-up tree automata is constructed out of the universal safety automaton.

Now, we describe their algorithm for computing the number of tree models whose tree bases are accepted by the bottom-up tree automaton. The algorithm starts at the initial states, which involves the initial conjectures for the number of expected repetitions of the initial annotation. Moreover, each initial state is mapped to the number of expected repetitions of the initial annotation. Then, it tracks each state in each possible transition. Here, a transition represents a parent state and its child states. This transition exists if, and only if, the initially guessed annotation was correct. For an annotation, there are initial guesses, which describe the number of loop back transitions of each leaf having this annotation.

In the k-th iteration, each accepting state q' of the bottom-up tree automaton $\mathcal{T}_{\#}$ is mapped to a function that defines for each annotation in the domain of $f_{q'}$ the number of possible loop combinations for the annotation in a tree accepted by q'. In the end, all possible loop combinations for each defined annotation are multiplied and, thereby, we obtain the number of tree models of the accepted base. Finally, they sum up the results for all accepting states.

The algorithm traverses the automaton k times resulting in a complexity of $\mathcal{O}(k).2^{2^{2\mathcal{O}(|\varphi|)}}$. Again, this approach is limited to formulas of small length, because of the threefold complexity. Since this approach is therefore impractical in practice, we develop two approximate counting algorithms for tree models with lower complexity. These approaches are presented in Chapter 5. Note that we do not develop an exact algorithm for the tree counting algorithm due to the high complexity of the tree counting problem.

4.2 Exact Model Counting

Before considering bounded model counting, we need to take a closer look at the problem of bounded model checking, in order to provide basic insights that can be reused for model counting. In this section, we present some background knowledge about the SAT encoding of the LTL model checking problem as well as an exact algorithm based on this technique.

4.2.1 Bounded Model Checking

Due to the fact, that SAT solvers have become very efficient, the authors in [2] aim to transfer the success of SAT solving to model checking. In model checking, we do only not want to check simple safety properties, e.g., how we can reach a goal state or whether it is reachable but we also want to check liveness properties and nested temporal properties. Bounded model checking can be used for all safety, liveness and nested temporal properties. In order to reduce LTL formulas to SAT formulas, we consider only a finite prefix of the formula unrolled to SAT. This prefix is restricted by some bound k, which, in case of word models, is the length of the base of the word models we want to check. Note that, if there is a back loop from the last state

of the prefix to any of the previous states, then this finite prefix still might represent an infinite path. By recalling the definition of word models, we can see that this is the case.

Definition 11 (SAT encoding of the LTL model checking problem) *Let φ be a LTL formula in PNF, k the bound, $l_i = true$ if $i \in \{0, ..., k-1\}$ and j is the state the back loop points to. Then we can reduce LTL to propositional logic with the following rules:*

1. $[p]_i = p_i$ *for* $i < k$
   $[p]_i = \bigvee_{j=0}^{k-1}(l_j \wedge p_j)$ *for* $i = k$

2. $[\neg p]_i = \neg p_i$ *for* $i < k$
   $[p]_i = \bigvee_{j=0}^{k-1}(l_j \wedge \neg p_j)$ *for* $i = k$

3. $[X\varphi']_i = [\varphi']_{i+1}$ *for* $i < k-1$
   $[X\varphi']_i = \bigvee_{j=0}^{k-2}(l_j \wedge [\varphi']_{j+1})$ *for* $i = k-1$

4. $[\varphi_1 U \varphi_2]_i = [\varphi_2]_i \vee ([\varphi_1]_i \wedge [\varphi_1 U \varphi_2]_{i+1})$ *for* $i < k$
   $[\varphi_1 U \varphi_2]_i = \bigvee_{j=0}^{k-1}(l_j \wedge \langle\varphi_1 U \varphi_2\rangle_j)$ *for* $i = k$
   $\langle\varphi_1 U \varphi_2\rangle_i = [\varphi_2]_i \vee ([\varphi_1]_i \wedge \langle\varphi_1 U \varphi_2\rangle_{i+1})$ *for* $i < k$
   $\langle\varphi_1 U \varphi_2\rangle_i = false$ *for* $i = k$

5. $[\varphi_1 R \varphi_2]_i = [\varphi_2]_i \wedge ([\varphi_1]_i \vee [\varphi_1 R \varphi_2]_{i+1})$ *for* $i < k$
   $[\varphi_1 R \varphi_2]_i = \bigvee_{j=0}^{k-1}(l_j \wedge \langle\varphi_1 R \varphi_2\rangle_j)$ *for* $i = k$
   $\langle\varphi_1 R \varphi_2\rangle_i = [\varphi_2]_i \wedge ([\varphi_1]_i \vee \langle\varphi_1 R \varphi_2\rangle_{i+1})$ *for* $i < k$
   $\langle\varphi_1 R \varphi_2\rangle_i = true$ *for* $i = k$

Definition 11 shows how this unrolling works. Basically, we express for every position in this word model which variables need to become true such that the LTL formula is satisfied in this position. For example for $\Box a$, in each position i, the variable a_i must be true. We start with $i = 0$ and unroll the formulas until $i = k - 1$ and then we start with unrolling every possible loop once.

Atomic proposition p: For atomic propositions, we transform p into a variable p_i by adding the current position i. When reaching the loop state, we iterate over all possible positions to loop to, connecting them by an $\vee$-operator. Each loop is represented by the $\wedge$-combination of the atomic proposition at the position we loop to and an unique variable l_j. We use l_j to take care that only one loop can be selected for each assignment.

Negated atomic proposition $\neg p$: The transformation for formulas of the type $\neg p$ works exactly the same way. The only difference is that we negate the retrieved variables.

Formulas that contain X-operator $X\varphi'$: For formulas of the type $X\varphi'$, we simply encode the formula φ' for the next position. Moreover, we do the same for expressing the loops as before. The only difference is that we compute the transformation of φ' for the next position instead of a variable expressing the atomic proposition or its negation.

Formulas that contain U-operator $\varphi_1 U \varphi_2$: For the formulas of the type $\varphi_1 U \varphi_2$, the SAT encoding at position i says the following: Either φ_2 is fulfilled at position i or φ_1 is fulfilled and the SAT encoding of the formula at the next position is fulfilled. When we start encoding the loop, we compute the chain of terms connected by an $\vee$-operator as before.When

encoding the looped until operator, however, we use mutual recursion in order to express the differences: the whole formula is encoded like before until we reach position $k-1$. After that, the formula is just encoded to false.

Formulas that contain R-operator $\varphi_1 R \varphi_2$: The encoding for the release operator is similar to the encoding for the until operator except that the SAT encoding says that φ_2 and either φ_1 or the SAT encoding of the formula at the next position are fulfilled. Moreover, the mutual recursion encodes the formula at the end of the loop to true.

Next, we need to transform our models into formulas. This is necessary, because we want to count models of a specific bound k and without this limitation, we would also count models with a lower bound or empty models. For word models this transformation is quite simple. For each position, we build a clause out of all atomic propositions and then, we connect these clauses with an $\wedge$-operator. This formula represents all possible word models for this length.
For tree models it is a little bit more complicated and we therefore build one formula for each possible tree model. Thereby, we build one formula for each path and connect them through an $\wedge$-operator. The formula for a path is build similar to the formula for word models, but instead of building a clause of all variables of that position, we build one clause for all variables that represent atomic propositions in the position and one clause for all variables that do not represent atomic propositions in that position. The latter clause only contains negated variables.

Finally, we connect both retrieved formulas through an $\wedge$-operator.

4.2.2 Counting Word Models without Loops

Now we are moving to our actual goal of model counting. For bounded model counting without loops, we start with transforming our LTL formula into PNF. After that, our formula does not longer contain any $\Box$-, $\Diamond$-, $\rightarrow$- or $\leftrightarrow$-operators. This is necessary since our bounded model checking rules are not defined for these operators.
Next, the transformation of the LTL formula into the SAT formula begins. Since we are not considering loops, we can simply transform the formulas with help of the rules presented in Definition 12. These rules are similar to the rules presented in Definition 11, but do not contain a case for loops.

Definition 12 (Model Counting via propositional Counting) *Let φ be a LTL formula in PNF, k the bound and we consider word models without loops. Then, we can reduce LTL to propositional logic with the following rules:*

1. $[p]_i = p_i$ *for* $i \leq k-1$
2. $[\neg p]_i = \neg p_i$ *for* $i \leq k-1$
3. $[X\varphi']_i = [\varphi']_{i+1}$ *for* $i \leq k-2$
4. $[\varphi_1 U \varphi_2]_i = [\varphi_2]_i \vee ([\varphi_1]_i \wedge [\varphi_1 U \varphi_2]_{i+1})$ *for* $i < k$
   $[\varphi_1 U \varphi_2]_i = false$ *for* $i = k$
5. $[\varphi_1 R \varphi_2]_i = [\varphi_2]_i \wedge ([\varphi_1]_i \vee [\varphi_1 R \varphi_2]_{i+1})$ *for* $i < k$
   $[\varphi_1 R \varphi_2]_i = true$ *for* $i = k$

This transformation works as expalained in the bounded model checking section, except that when reaching the end of the model, we do not enter the loop case. In general, for bounded model counting, we would have to encode our model into a SAT formula as well. If we would regard empty word models or word models with a length smaller than the bound, this would not be necessary. Therefore, we transform the word model into a formula by connecting all possible variables of position zero with an $\vee$-operator and then do the same for all other positions in the word model. Finally, we connect all these disjunctive formulas with several $\wedge$-operators. For example, the formula for a word model with bound 2 and 2 atomic propositions would be $(a_0 \vee b_0) \wedge (a_1 \vee b_1)$. For model checking, we would have to encode the model we want to check, for counting, we need to encode every possible structure of our word models.

After concatenating the formula generated out of the word model with the formula received via bounded model checking through an $\wedge$-operator, we can simply check the satisfiability of the formula and add one for each solution per word model that satisfies the formula. Furthermore, we guarantee this by taking all variables (also those that do not appear in the formula) into account.
As mentioned before, we chose this kind of approach in order to make use of the results of SAT counting for LTL counting. Therefore, we choose three good SAT counters to evaluate the model count. These three SAT counters are the exact counter Relsat [1] and the approximate counters ApproxCount [18] and ApproxMC [4]. All of them can only process SAT formulas delivered in DIMACS format. In order to convert a SAT formula into DIMACS format, the SAT formula needs to be in Clausal Normal Form (CNF).

After converting the formula into CNF, our next step is transforming the formula into DIMACS format. In DIMACS format, all variables are represented by numbers. A negative number means the variable is negated. As an input, the DIMACS format gets the number of clauses and the number of variables. First of all, we count the number of clauses we have. Since our formula is in CNF, we can simply count the number of $\wedge$-operators and add one. The number of variables can be computed by multiplying the number of atomic propositions with the bound k. When we generate a new literal, this literal gets a unique number. For each clause, we add the arrays of its components to the list that contains all clauses. For each literal, we generate a new array and concatenate this array with the arrays of the literals it is connected to via an $\vee$-operator. For each negated literal, we negate its value in its array.
Finally, we can write the DIMACS-formated formula into a document and hand it over to Relsat, which determines the number of solutions.

4.2.3 Counting Word Models with Loops

This approach is quite similar to the approach presented in the previous section. However, there are a few differences. Since we are considering word models with loops, we use the formulas presented for SAT encoding in Definition 11 instead of the formulas presented in Definition 12 in order to transform our LTL formula into a SAT formula. The tranformation works just as explained in the bounded model checking section. Moreover, we need to build k SAT formulas out of our LTL formula, one for each possible loop for the word models. Therefore, we add several variables that denote the different loop-back transitions. These variables are put into clauses that restrict the use of loops per assignment to exactly one.

4.2.4 Relsat

Now, we are going to explain the exact SAT counter Relsat [1]. Relsat was developed to demonstrate the practical utility of CSP look-back techniques. It is a look-back enhanced algorithm that is used to solve large SAT instances derived from real-world problems in planning. These look-back techniques are used to exploit information about searches that has already taken place. They present 3 different approaches for SAT counting, one of them being Relsat.

All these procedures are based on the Davis-Putnam procedure, working as follows: If a contradiction is found while applying the Davis-Putnam unit-propagation procedure, usually a failure is reported and backtracking is initiated. For selecting the branch variable, a heuristic returns the next variable that should be valued and if neither truth value works, a failure is reported.

However, their algorithms use a modified version of the David-Putnam procedure. In unit-propagation, a single literal λ, derived from a unit clause, is added to the set σ. Afterwards, the CNF is simplified by removing all clauses that contain λ and shortening those that contain $\neg\lambda$ through resolution. Selecting the branch variables also works different. If there is no binary clause, than a branch variable is selected at random. If there is a binary clause, then a score is assigned to each variable x that appears in a binary clause. This score is computed from the number of negated and not negated appearances of x. Next, all candidates that scored within the best 20% of all scores are put into a candidate set. The size of this candidate set is restricted to a maximum of 10. If there are more candidates than space in the candidate set, some of the candidates are removed at random. If this technique only gathered one candidate, they select this candidate as their branch variable. Otherwise, they re-score their candidates. First, the positive and negative values are computed as retrieved from unit propagation until a contradiction is found. Next, they select the variable that triggered the contradiction as the branch variable. If no contradiction is found, the branch variable is selected at random from the best 10% of scored variables and $true$ is assigned to the branch variable. They support their algorithm by implementing conflict directed backjumping (CBJ). Thereby, the unit-propagation method maintains a pointer to the clause in the input file that is used for excluding a specific assignment from consideration. A contradiction occurs when both truth values of a variable are excluded. Moreover, they apply restricted learning schemes. For Relsat, relevance-bounded learning is applied.

In their experiments Relsat performs good. It can compute 4 out of 6 scheduling instances with 100% success rate and outperforms other state-of-the-art SAT solvers.

4.3 Implementation

In this section, we detail the implementation of our exact approaches for word models, having a more detailed description on the actual data structures we used. Our programs are written in Java.

4.3.1 Bounded Model Counting for Word Models

The implementation of this approach is based on the Bounded Model Checking approach for LTL formulas known from the work of Clarke et al. [2]. We take both infinite and finite word

models into account and bound them by the length of the base of the word model k. As input the program gets the formula, the bound and the number of atomic propositions.

Bounded Model Counting without Loops

First of all, the atomic propositions are generated. In our case, they are represented through letters. For example, if we enter five as the number of atomic propositions, our generator generates the following set $AP = \{a, b, c, d, e\}$. Before applying the formulas for the actual approach (recall Definition 12), we parse the String that represents the LTL formula and transform the formula into positive normal form (PNF). This transformation is done by the tool presented in the approach form Giannakopoulou et al. [6]. Note that, for word models, we only use the translation into PNF and not the translation into Büchi automata. After that we are ready to apply the bounded model checking formulas. By applying these formulas, we convert all our atomic propositions into several variables. For example the atomic proposition a is transformed into the variables $a_1, a_2, \ldots, a_k$, if our base has length k.

Furthermore, in order to convert the formula into a machine readable format, we first transform it into CNF and then into the DIMACS format that serves as an input to our SAT counters. For transforming the formula into the DIMACS format, we count the number of variables and the number of clauses we have. Moreover, we assign a value to each variable and for every clause we put all variables that it contains into a list. Note that negated variables are depicted as negative numbers.
Since our sat counters cannot be embedded into our Java code, we have to write the DIMACS encoding of the formula into a file, before calling them as an external program and redirecting their output back to our Java program.

As an exact SAT counter, we decided to use Relsat [1]. We chose Relsat because of its good results. By only considering a few variables, it can also outperform our approximate SAT counters.

As approximate SAT counters, we chose ApproxCount [18] and ApproxMc [4]. The main reason for choosing ApproxCount is that ApproxCount delivers results that only differ slightly from the exact number. Our main reason for choosing ApproxMC is that it is one of the latest SAT counting approaches that promises to deal with formulas presented in CNF much better than, for example, ApproxCount.

Bounded Model Counting with Loops

The approach with loops is implemented almost exactly the same way than without loops. The difference is the algorithm we use to implement the tranformation from LTL to propositional logic. We use the formulas presented in Definition 11 instead of using the formulas in Definition 12.

4.3.2 Counting Word Models of LTL

As input the program gets the universal safety automaton built out of the specification, the bound k represented by an integer value and the number of atomic propositions represented by

an integer value. The universal safety automaton is represented by an array of long values that represents the states, a long value that represents the initial state, a map that maps a state to a set of atomic propositions and another state, which represents the transition function δ and a list of strings that represents the alphabet of the automaton.
Next, the universal safety automaton is transformed into the word automaton. The components of the automaton are represented by a list of conjecture states and a list of strings that defines the alphabet of the automaton. The conjecture states are represented by the conjecture set and the tracking sets as well as a counter. Thereby, both sets are represented by a HashSet and an array of HashSets. The transition function is represented by a method that computes the transitions on input of the state the transition starts and the transition symbol.
Finally, the counting algorithm is implemented as described in the corresponding section.

CHAPTER 5

APPROXIMATE MODEL COUNTING

In this chapter, we explain both, approximate SAT approaches using bounded model counting as well as our approach using Monte Carlo. We start with explaining the SAT approaches first and continue with an explanation of the Monte Carlo method and our algorithm based on this method.

5.1 Word Models

In this section, we take a closer look at our approximate approaches for word models.

5.1.1 Bounded Model Counting for Word Models

The first approach that uses approximate techniques for model counting is closely related to the exact bounded model counting approach presented in Chapter 4. Taking advantage of approximate SAT counters, we encode the problem in SAT formulas. The first approximative SAT counter we use is ApproxCount [18] and the second one is ApproxMC [4].

ApproxCount

ApproxCount is a tool for approximating the number of satisfying assignments or models of a SAT formula developed by Wei et al. [18]. ApproxCount can also provide a good estimate for those formulas that cannot be handled by most current counters.

While most SAT counters are based on backtracking techniques like DPLL, ApproxCount is based on a biased random walk strategy and uses an algorithm called SampleSat [17] that draws near-uniform samples from the solution space of a formula in propositional logic. The advantage of such a sampling approach is that it enables runtime control. They can simply obtain different maximum runtimes by changing how many samples are drawn in each iteration. Note that, the more samples are drawn, the more exact the count will be. ApproxCount performs as many iterations as variables are considered and SampleSat is called in each iteration.
Algorithm 2 explains how ApproxCount determines its count for a formula φ. First of all, K samples are drawn from the solution space of φ. Next, a variable x, that occurs in φ, is chosen at random. For all samples, it is checked whether x is assigned $True$ or $False$ more often and allocate all x in φ to the value it is assigned in the most samples. Finally, they compute the

Algorithm 2 ApproxCount Algorithm

```
while φ! = empty do
    Draw K samples from the solution space of φ
    x := choose a variable in φ at random
    for all K samples do
        if #(x = True) > #(x = False) then
            φ :=Unitprop(φ, x = True)
            multiplier M_x = K/#(x = True)
        else
            φ :=Unitprop(φ, x = False)
            multiplier M_x = K/#(x = False)
        end if
    end for
    return product of all multipliers
end while
```

multipliers and return their product.

Wei et al. [18] evaluated ApproxCount on several domains. In general, they received a good estimate for all formulas and ApproxCount is able to extend the range of formulas for which the number of models that satisfy the formula can be computed. Moreover, their experiments showed that ApproxCount has a low error rate for random formulas and that it approximates the true count for structured formulas within a factor of 2 or better. Unlike exact counters, ApproxCount's runtime does not grow exponentially. Its runtime is upper bounded by the product of the number of variables, the number of solutions drawn in each iteration and the time needed to draw a solution. Therefore, ApproxCount has a polynomial runtime.

ApproxMC

ApproxMC [4] is the first scalable approximate model counter for CNF formulas. It counts the number of satisfying assignments within a given tolerance ϵ and a confidence level $1 - \delta$. By running ApproxMC with high confidence, it can deliver results that are very close to the exact counts.

ApproxMC begins with randomly partitioning the set of models into small cells. In order to determine whether a cell is small, it checks if a random cell is non-empty and has less than $pivot$ elements, where $pivot$ is a threshold that depends on the tolerance ϵ. If the cell is not small, it partitions the cells into twice as many cells. This process is repeated until it either finds a small cell or the number of cells is larger than $\frac{2^{n+1}}{pivot}$. If no small cell is found, a counting failure is reported.

ApproxMC works with help of the core engine ApproxMCCore. The ApproxMC algorithm, shown in Algorithm 3, invokes ApproxMCCore many times and returns the median of the computed counts that do not result in a counting failure. As an input, the ApproxMC algorithm gets a CNF formula φ, a tolerance ϵ, and a value δ that is used to compute the confidence level. The algorithm computes the value of $pivot$ that is used to determine the size of a small cell and a

parameter $t \geq 1$, that depends on δ. t is used to determine how often we invoke ApproxMCCore.

Algorithm 3 ApproxMC($\varphi, \epsilon, \delta$)

```
counter = 0, C = emptyList();
pivot = 2 * ⌈3e^(1/2)(1 + 1/ε)^2⌉;
t = ⌈35 log2(3/δ)⌉;
repeat
    c = ApproxMCCore(φ, pivot);
    counter = counter + 1;
    if (c does not return a counting failure) then
        C.add(c);
    end if
until (counter > t)
finalCount = FindMedian(C);
return finalCount;
```

The ApproxMCCore algorithm presented in Algorithm 4 takes as input a CNF formula φ and our threshold $pivot$ and returns an ϵ-approximate estimate of the model count of φ. This algorithm has access to a function BoundedSAT that takes as input the propositional formula φ', which is a conjunction of a CNF formula and xor-constraints, as well as a threshold $v \geq 0$. It returns a set S of models of φ' such that $|S| = min(v, \#\varphi')$. If the model count is not larger than the pivot element, the exact count is returned. If it is, the space of all models of φ is partitioned, a random cell is chosen and it is checked whether it is small. We continue with checking cells until a small cell is found or we generate more cells than permitted. Finally, for all small cells, the size is scaled by the number of generated cells in order to compute an estimated model count.

In their evaluation, ApproxMC was compared to an exact solver and state-of-the-art approximate bounded counters. At the beginning the exact solver performed better than ApproxMC, but when the problem-size increased, it timed out. In general, the results of ApproxMC were near the exact count and it computed better results than the bounded model counters.

Our experimental results for all exact and approximate SAT encoding approaches are presented in Chapter 6.

5.1.2 Counting via Monte Carlo

Our next approach is a LTL counting algorithm based on the Monte Carlo technique. In order to check the satisfiability, we use model checking a path [11]. Again, we introduce an approach for word models with and without loops. However, we start with explaining the Monte Carlo method in detail.

Monte Carlo Method

The Monte Carlo method is a technique that is used to solve analytically not solvable or only hardly solvable problems with the help of random experiments. These random experiments are run many times until we receive a meaningful result.

Algorithm 4 ApproxMCCore($\varphi, pivot$)

Assume that $z_1, \ldots, z_n$ are the variables of φ;
S = BoundedSAT($\varphi, pivot + 1$);
if $|S| \leq pivot$ **then**
  return $|S|$;
else
  $l = \lfloor log_2(pivot) \rfloor - 1$; $i = l - 1$;
  repeat
    $i = i + 1$;
    Choose h at random from a set of Hash functions;
    Choose α at random from $\{0,1\}^{i-l}$;
    S = BoundedSAT($\varphi \wedge (h(z_1, \ldots, z_n) = \alpha), pivot + 1$);
  until $(1 \leq |S| \leq pivot) \vee (i = n)$
  if $((|S| > pivot) \vee (|S| = 0))$ **then**
    return counting failure;
  else
    return $|S| * 2^{i-l}$;
  end if
end if

We start with a simple example in order to explain the Monte Carlo method properly:

Example 1 (Calculating an area of a irregular region [9, 15]) *The Monte Carlo method can be used to determine the area of an irregular region $\mathcal{X}^*$. To compute this area, we insert our irregular region into a regular one. In this case, we want to regard a rectangle as you shown in Figure 5.1.*
After that, we apply the following rules:

1. *Generate a random sample $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ uniformly distributed over the regular region $\mathcal{X}$.*
2. *Estimate the desired area $|\mathcal{X}^*|$ as*

$$\widehat{\mathcal{X}^*} = |\mathcal{X}| \frac{1}{N} \sum_{k=1}^{N} I_{\{X_k \in \mathcal{X}^*\}},$$

where $I_{\{X_k \in \mathcal{X}^*\}}$ denotes the indicator of the event $\{X_k \in \mathcal{X}^*\}$ and we accept the generated point X_k, if $X_k \in \mathcal{X}^*$ and reject it otherwise.

Unsurprisingly, we can use the Monte Carlo method also for estimating other things than areas. In this thesis, we use it to generate assignments for SAT formulas and also to generate models for LTL formulas. To apply the Monte Carlo method to model counting, we use the rules presented in Definition 13. More details then follow in the following sections that present the specific applications for word and tree models respectively.

Definition 13 (Monte Carlo for SAT and LTL formulas) *We define the Monte Carlo method for both SAT and LTL counting:*

1. *Monte Carlo for the SAT counting problem*

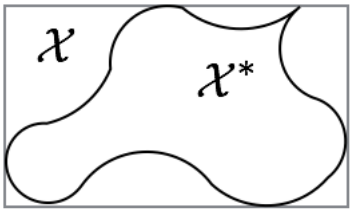


Figure 5.1: Illustration of the area we want to determine [15]

(a) *Generate a set of assignments $X_1, \ldots, X_N$ uniformly at random, where $X_i \in \mathcal{X}$ and $i \in \{1, \ldots, N\}$ and $\mathcal{X}$ is the set of all possible assignments for the formula.*

(b) *Estimate the desired number of satisfying assignments with help of the following formula:*

$$\widehat{\mathcal{X}^*} = |\mathcal{X}| \frac{1}{N} \sum_{k=1}^{N} |X_k|$$

,where N is the number of generated assignments, and $|X_k| = 1$, if X_k satisfies the formula.

2. *Monte Carlo for the LTL counting problem*

(a) *Generate several models $X_1, \ldots, X_N$ of length l uniformly at random, where $X_i \in \mathcal{X}$, $i \in \{1, \ldots, N\}$ and $\mathcal{X}$ is the set of all possible models of length l over the set of atomic propositions, the formula is defined over.*

(b) *Estimate the desired number of satisfied models with help of the following formula:*

$$\widehat{\mathcal{X}^*} = |\mathcal{X}| \frac{1}{N} \sum_{k=1}^{N} |X_k|,$$

where N is the number of generated models, and $|X_k| = 1$, if X_k satisfies the formula.

Model Counting via Monte Carlo without Loops

In a first step, we need to convert our formula into PNF.
Next, we start generating words of length k. We choose k sets of atomic propositions at random and concatenate them to a word as described in Algorithm 5. Thereby, we choose a random number $r \leq |AP| - 1$ and after that concatenate the r-th atomic proposition to the word. This procedure is repeated until we have a significant number of word models.

After creating all necessary word models, we begin with determining whether they satisfy our formula or not. As mentioned before, we use the method presented by Markey et al. [11]. We first split the formula into parts. Out of the formula aUb we receive the formulas a, b and aUb.

Algorithm 5 generateWordBase(AP)

```
word = null;
for (i = 0, i ≤ k − 1, i + +) do
    r = new random ∧r ≤ |AP| − 1;
    word.concat(AP(r));
end for
return word;
```

Then, we build a table for each word model, that assigns each subformula and position in the word model a truth value. An example for such a table is shown below. An entry in the table is true, if the respective formula holds beginning at the current position. Otherwise the entry is assigned false.

Example 2 (CTL Model Checking for the formula $\neg aUb$ and the word $bbaacc$)

	b	*b*	*a*	*a*	*c*	*c*
a	*false*	*false*	*true*	*true*	*false*	*false*
b	*true*	*true*	*false*	*false*	*false*	*false*
aUb	*true*	*true*	*false*	*false*	*false*	*false*
$\neg aUb$	*false*	*false*	*true*	*true*	*true*	*true*

The table is build as described below. Note that, while processing a formula, we first process its subformulas and store the results in the table. The algorithm works as follows:

Atomic proposition p: The atomic proposition is the lowest level of all our subformulas. For each position i in the word model, we check whether the i-th atomic proposition in the word model is equal to p.

Negated atomic proposition $\neg p$: For each position, we look up the result for the atomic proposition p in the table and negate the result.

Formula with X-operators $X\varphi$: For each position i, we look up the result for φ at position $i + 1$ and write the result in the table at position i.

Disjunction $\varphi_1 \vee \varphi_2$: For each position i, we look up the results for φ_1 and φ_2 in the table. If at least one of the results is true, our result for this position is true. If not, the result is false.

Conjunction $\varphi_1 \wedge \varphi_2$: For each position i, we look up the results for φ_1 and φ_2 in the table. If at both results are true, our result for this position is true. If not, the result is false.

Formula with U-operator $\varphi_1 U \varphi_2$: Starting at the last position of the word model, for each position i, we look up the results for φ_1 and φ_2 in the table. Until we reach a position where the result for φ_2 in the table is true, we write the value false into all positions of the table and assign true to this position. After reaching this position, we continue checking the results of φ_1 and φ_2 in the table and if at least one of the results is true, our result for this position is true.

Formula with R-operator $\varphi_1 R \varphi_2$: For release, we also start from the last position in the word model, looking up the truth values of φ_1 and φ_2. For release to be true from a given

position i, either φ_2 holds at all positions $j \geq i$ until the end or φ_2 holds until some position $j > i$ at which both φ_1 and φ_2 hold.

Afterwards, we take a look at the first value of the boolean array stored for the whole formula. Is this value $true$, we increase our counter, because this means that the formula holds from the beginning of the word model. Finally, we return the value of the counter divided by the number of models we created.

Model Counting via Monte Carlo with Loops

In this section, we consider word models with loops. Basically our algorithm presented in the previous section remains the same. The only thing we need to change, is how we generate our models. Until now, we only generated a base of a word model. Therefore, we need to add the loop-back transition to our models.
In Algorithm 6, we describe how to generate these word models. At the beginning, we generate our word base as shown in Algorithm 5. Afterwards, we choose a random value l that marks the beginning of the loop-back transition. Next, we memorize the part of the word from l to $k-1$ in the variable $loop$ and concatenate the base with $loop$. Finally, we process our word models with the same algorithms described in the previous section.

Algorithm 6 generateWordModel(AP)

```
word = null;
for (i = 0, i ≤ k − 1, i++) do
    r = new random∧r ≤ |AP| − 1;
    word.concat(AP(r));
end for
l = new random ∧r ≤ |AP| − 1;
loop = null
for (i = l, i ≤ k − 1, i++) do
    w = word.get(i);
    loop.concat(w);
end for
return word.concat(loop);
```

5.2 Tree Models

In this section, we concentrate on our approaches for tree models. Note that we do consider bisimilar trees as being equal and therefore count only one of them.

5.2.1 Bounded Model Counting for Tree Models

First, we present two bounded model counting approaches, one that takes does not loops into account and one that does. Both of them generate a certain number of trees at random and then use Relsat [1] for satisfiability checking.

Bounded Model Counting without Loops

Our algorithm gets as input the number of atomic propositions, the depth of the trees that should be generated and the number of trees, we want to generate. We again start with parsing our formula and transforming it into PNF.

Next, we start generating the trees. A tree model consists of a root node, child nodes and several leaf nodes. We start with the set of atomic propositions AP and the depth of the tree k as input and create an empty tree of depth k. Then, we build up the tree by creating children for each depth-level while maintaining a list of path labels that is used to ensure that we do not have duplicate paths in the tree.
When constructing the actual tree, we start generating the node on depth level zero, i.e., the root node. To assign a label to a node, we first determine the size of the set of atomic propositions for this node at random between 1 and k. Then, we fill the set with randomly chosen atomic propositions, while ensuring that no atomic proposition is chosen twice. Each node also stores information about its depth, its parent and its children.
After generating the root node, we continue with depth-level 1, i.e., the root nodes children. All generated children on the current depth-level are stored in a list $futureParents$, so that we, later on, can assign the correct parent to their child nodes. More specifically, beginning from depth-level 1, we repeat the following process for each parent node p:

1. Generate a list $children$ that will contain all child nodes for this parent.
2. Determine the number of children we want to generate for this parent at random between 1 and k.
3. Generate the child nodes:
   (a) Determine labels for the child node the same way as described for the root node.
   (b) Generate a new node and correctly set the information about its depth and its parent. Then add this node to the list of child nodes of p, to the tree and to the list of $futureParents$.
   (c) Update the list of path labels.
4. Build a reference to each child in the parent node p.

We repeat this process until we reach the maximum depth level k. Then, we eliminate duplicate paths according to the list of path labels and return the tree.

Note that we bound the tree not only by depth, but also by the degree. Since we only want to consider unique paths, it would slow down the performance of the program enormously. In order to avoid this, we only consider nodes with maximum of $|AP|$ child nodes.

Now we can start with our actual counting algorithm. This algorithm is based on the model checking algorithm that is repeated for each tree model. After each run is finished, we increase a counter when the tree model satisfies the formula.

To check if a model satisfies the formula, we first convert our LTL formula into a SAT formula. This can be done similar to the approach presented for word models, but since we are considering tree models and tree models are a composition of word models, we need to make some

changes in the conversion. A tree model fulfills a formula, when all its parts fulfill the formula and we need to differentiate between atomic propositions used in several paths. Appending the depth-level to atomic propositions does not suffice in this case, because the same atomic proposition can occur multiple times in the labels of the given depth-level. Consequently, we assign each node a unique id. Furthermore, our algorithm converts each path separately, so that we also need to get all paths of a tree. In Algorithm 7 and 8, we show how we obtain the paths.

Algorithm 7 getPaths()

List $paths$;
Array $pathSoFar$;
getPathsById(getRoot(), $pathSoFar$, $paths$)
return $paths$;

Algorithm 7 initializes the gathering of all the paths of the tree and returns the list of all paths when Algorithm 8 is finished for all nodes.

Algorithm 8 getPaths(Node n, $pathSoFar$, $paths$)

Array $curPath$ = copyOf($pathSoFar$, $pathSoFar.length$);
$curPath[n.getDepth()]$ = n.getId();
if (!n.isLeaf()) **then**
  for (int i = 0; $i < n$.getChildNumber(); i++) **do**
    getPathsById(n.getChild(i), $curPath$, $paths$);
  end for
else
  n.addPath($curPath$);
  $paths$.add($curPath$);
end if

Algorithm 8 is called for each node and sets the unique id of each node into its corresponding path arrays. Then it calls itself for all the child nodes of the current node n and upon reaching a leaf node, it adds the path $curPath$ to the list of all paths $paths$.

After that, we can start transforming our LTL formula into a SAT formula. The only difference from the conversion for word models is that it also gets the current path as an input. This conversion is repeated for every path of the tree model and the formulas are connected with an $\wedge$-operator, resulting in a SAT formula consisting of the subformulas for each path.
Next, we convert our tree model into a formula. In contrast to the bounded model counting approach for word models, where we encoded all possible word models into propositional logic, here we have a specific tree model that needs to be represented. The conversion of a tree model into a SAT formula is shown in Algorithm 9.

In Algorithm 9, we convert a tree into a SAT formula by traversing the tree using DFS recursion. For each node n, we build a formula $\varphi_n = \bigwedge_{p \in o(n)} p_{id(n)} \wedge \bigwedge_{p \in AP \setminus o(n)} \neg p_{id(n)}$. φ_n contains all atomic propositions in the label (concatenated with the nodes unique id) as positive literals and all atomic propositions not occurring in the label (also concatenated with the nodes id) as

Algorithm 9 convertTree(Node n, AP)

```
formula φ;
List children;
φ' = new Literal(n.getName(0));
for (int i = 1; i < n.getNames().size();i++) do
    φ' = φ'∧ new Literal(n.getName(i));
end for
HasSet not = AP;
not.remove(n.getNames());
for (String ne : not ) do
    φ' = φ'∧ ! new Literal(ne + n.ID)
end for
if (!n.isLeaf()) then
    Node child = n.getChild(0);
    φ'' = convertTree(child, AP);
    for (int i = 1; i < n.getNames().size();i++) do
        φ'' = φ''∧ convertTree(n.getChild(i), AP);
    end for
    return φ' ∧ φ'';
else
    return φ';
end if
```

negative literals. All literals are connected by an $\wedge$-operator. In the end, the algorithm returns a formula that combines all φ_n into one formula using the $\wedge$-operator.

After that, we connect the SAT formula obtained by converting the tree with the SAT formula obtained by converting the LTL formula through an $\wedge$-operator. This formula is then converted into CNF and afterwards converted into DIMACS format. Finally, we feed the DIMACS formula into Relsat [1] and the satisfiability is checked. If the formula is satisfiable, we increase the counter.
This is repeated for each randomly generated tree model and at the end we output the counter.

Bounded Model Counting with Loops

The algorithm in this section is similar to the one presented in the previous section. The main difference occurs when transforming the LTL formula into the SAT formula, because we need to take loops into consideration. For each path in the tree, there are several loop-back transitions possible. We therefore convert the formula as for word models with loops and modify this algorithm according to the changes for tree models we presented in the previous section.

We again use special loop literals, however, now we need them to ensure that at least one loop per path (instead of exactly one for word models) is chosen for an assignment. Besides leveraging the unique node id's to determine a literal's position in the tree as seen before, the loop literals must be annotated with a path number to distinguish between the loops of different paths.
For counting, we connect all terms, each representing a path in the tree, through an $\wedge$-operators

and call Relsat on the computed term. This time, Relsat does not only return whether the base satisfies the formula or not, it also computes how many models of that base satisfy the formula. At the end, we return the sum of these computations.

5.2.2 Model Counting via Monte Carlo

In this section, we present an automata based approximate tree counting algorithm. The algorithm is based on the automata based approach proposed by Vardi et al. [16] to check whether a model fulfills the formula.For our approach, the respective models are generated using Monte Carlo. Note that we only consider tree models and not their bases. Figure 5.2 gives an overview of this technique.

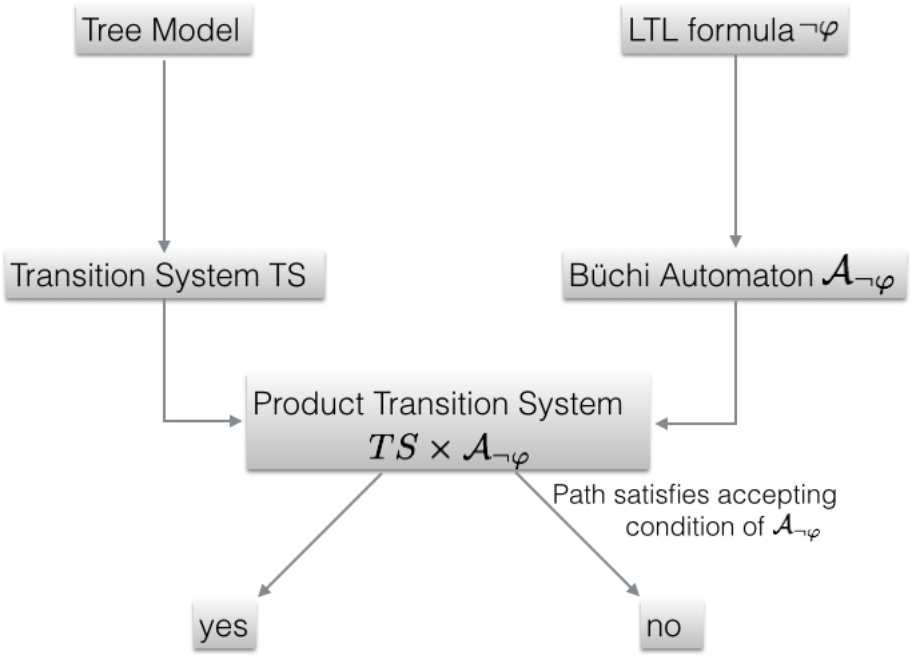


Figure 5.2: Automata Based Model Checking [16]

As an input, this algorithm gets the formula φ, the depth of the tree model k, the number of atomic propositions $|AP|$ and the number of tree models that should be generated.

For this approach, the tree models are generated as before. However, instead of encoding the loops in a formula, we now have to explicitly add them in our tree representation, because we are now considering tree models and not their bases. Recall that for tree models, each leaf can have several loop-back transitions. Therefore, for each leaf node, we first randomly select the number of loop-back transitions between 1 and k. Then, we randomly choose the actual transitions and store this information in the tree.

In order to check whether the tree model satisfies the formula, we rely on the fact that each LTL formula can be represented by an NBA. Basically, instead of directly checking if the tree satisfies the formula, we equivalently check if the product transitions system of the tree model and

the NBA of the negated formula does not contain a path satisfying the accepting condition of the NBA. To this end, we first parse $\neg\varphi$ and translate it into a Büchi automaton as presented in the approach of Giannakopoulou et al. [6]. Then, for each tree, we build the product transition system of the automaton and the tree as presented in Definition 14 and in Algorithm 10.

Definition 14 (Product of a Transition System TS and a Büchi Automaton $\mathcal{A}$) *Let $TS = (S, s_0, \tau, o)$ be an AP-labeled Υ-transition system without terminal states and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ a non-blocking NBA. Then, $TS \times \mathcal{A}$ is the following transition system:*

$$TS \times \mathcal{A} = (S \times Q, \tau', s_0', AP', o')$$

where

1. *τ' is the smallest relation defined by*

$$\frac{s \to t \wedge q \xrightarrow{o(t)} p}{\langle s, q\rangle \xrightarrow{\tau} \langle t, p\rangle}$$

2. *$s_0' = \{\langle s_0, q\rangle \mid s_0 \wedge \exists q_0 \in Q_0 . q_0 \xrightarrow{o(s_0)} q\}$.*
3. *$AP' = Q$.*
4. *$\tau' : S \times Q \to 2^Q$ is given by $\tau'(\langle s, q\rangle) = \{q\}$.*

As input, Algorithm 10 gets the tree t and the automaton b, we want to build the product of. Next, we access both the root node of the tree s_0 and the initial state of the automaton q_0 and compare whether the labels of the node and the labels of each of the outgoing edges are the same. If this is the case for one of the outgoing edges, we build a new initial node $\langle s_0, q\rangle$ for the transition system, where q is the state in the automaton the edge points to. We continue with checking the other edges of q_0 and after gathering all initial states, we return the transition system. Note that all other elements of the formal definition (including the transitions) can be computed on-the-fly. Therefore we do not precompute them.

An example of such a product transition system is shown in Figure 5.3.

Having the product transition system, we can start with the model checking process. In order to determine whether the accepting condition is fulfilled, we use Algorithms 11, 12 and 13. In these algorithms, we basically perform some nested depth-first search (DFS). We perform two kinds of depth first search, outer and inner DFS. The outer DFS is shown in Algorithm 11 and it explores all reachable states that are not accepted. The inner DFS is shown in Algorithm 13 and seeks backwards edges that lead to a state found in the outer DFS.
Starting with initializing the set of states visited in the outer DFS R, the stack for the outer DFS U, the set of visited states in the inner DFS T and the stack containing for the inner DFS V, we explore all states in the outer DFS, we did not visit yet. If we find a state we did not visit yet, we start with the inner DFS as shown in Algorithm 12 and change the value of the variable $cyclefound$ when a contradiction is found.

Algorithm 10 buildProduct(Büchi Automata b, Tree t)

```
Node s_0 = t.rootNode;
State q_0 = b.getInitialState;
List initialStatesTS;
labels = s_0.getLables;
for (edge : q_0.getOutgoingEdges) do
  if TS.checkGuard(labels, edge) then
    (*is true if the labels of the tree node are the same as the ones for the edge of the automaton*)
    State q = edge.getNextState;
    initialState.add(s_0, q);
  end if
end for
return new TS(initialStates);
```

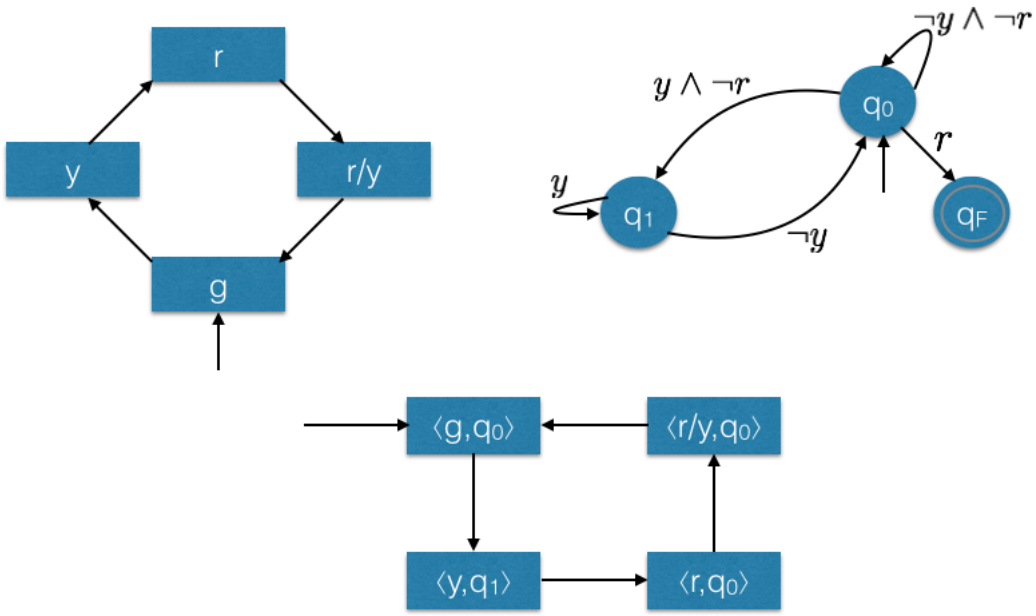


Figure 5.3: a transition system, a Büchi automaton and their product transition system [16]

Algorithm 11 persitencePropertyCheck(Transition system TS)

```
HashSet R;
Stack U;
HashSet T;
Stack V;
boolean cyclefound = false;
while (¬R.containsAll(TS.getInitialStates())∧¬cyclefound) do
  Node s;
  for (Node i : TS.getInitialStates()) do
    if (¬R.contains(i)) then
      s = p;
      break;
    end if
  end for
  cyclefound =reachableCycle(s, R, U, T, V);
end while
return ¬cyclefound;
```

As input, Algorithm 12 gets the node s we are visiting, the set R of visited states in the outer DFS, the stack U for the outer DFS, the set T of visited states in the inner DFS and the stack V for the inner DFS. First, we push the node s on the stack U and add it to the list R. Then we check whether R contains all successors of the state $s' \in U$. Is this not the case, then we push the unvisited successors of s' onto the stack U and add them to R. If it does contain all successors of $s' \in U$, then we pop s from U and if it is not accepted then we call the inner DFS, namely Algorithm 13.

As input, Algorithm 13 gets the node s, the stack V for the inner DFS and the set T of visited states in the inner DFS. We first push the node s on the stack V and add it to the set T. Then we repeat the following procedure until either $cyclefound$ is true or V is empty.

1. Take the top element s' of V.

2. If s is a successor of s', then we found a cycle and set $cyclefound$ to true. Furthermore, we push s on the stack.

3. If s is not a successor of s' and the set of successors of s without the set T is not empty, we push an unvisited successor s'' of s' on V and add it to T.

4. If the set of successors of s without the set T is empty, we pop V.

Finally, we return the actual value of $cyclefound$.

If these algorithms say that the property holds, we increase a counter. Finally we return the counter as the result of our model counting algorithm.

Algorithm 12 reachableCycle(Node s, Set R, Stack U, Set T, Stack V)

```
U.push(s);
R.add(s);
repeat
  s' = peek(U);
  Node s_1;
  if (Node i : s.getNext()) then
    if (¬R.contains(i)) then
      s_1 = i;
      break;
    end if
    U.push(s_1);
    R.add(s_1)
  else
    U.pop();
    if (s'.isNotAccepting()) then
      cyclefound = cycleCheck(s', T, V)
    end if
  end if
until (U.isEmpty() ∨cyclefound)
return cyclefound;
```

Algorithm 13 checkCycle(Node s, Set T, Stack V)

```
cyclefound = false;
V.push(s);
T.add(s);
repeat
  s' = peek(V);
  if (s'.getNext.contains(s)) then
    cyclefound = true;
    V.push(s);
  else
    if (¬T.containsAll(s'.getNext())) then
      Node s'';
      for (Node i : s'.getNext()) do
        if (¬T.contains(i)) then
          s'' = i;
          break;
        end if
      end for
    else
      V.pop();
    end if
  end if
until (V.isEmpty() ∨cyclefound)
return cyclefound;
```

5.3 Implementation

In this section, we describe the Implementation of all approximate approaches for both word and tree models.

5.3.1 Word Models

Regarding word models, we implemented two different approximate approaches. The first one is a Bounded Model Counting [2] approach that counts models both exact and approximate and the second one is an LTL counting approach that uses Monte Carlo. Since we already described the implementation for the first approach in the previous chapter, we will only describe the Monte Carlo algorithm.

Counting Word Models via Monte Carlo

In order to make LTL model counting faster than in the previous section, we chose to implement an approximate LTL model counter. As input the program gets the LTL formula φ, the number of word models k it should generate, the number of atomic propositions $|AP|$ we consider and the length of each base of each word model.

Counting Word Models via Monte Carlo without Loops Just like in the bounded model counting approach, the program starts with converting the formula into PNF with help of the tool presented by Giannakopoulou et al. [6] and generates the atomic propositions as explained above. Next, the program starts generating word models and puts them into a list. These word models are generated by choosing k sets of atomic propositions. To this end, the atomic propositions chosen after i random choices is put at the i-th position in the string array.

After that, we start checking whether the word models satisfy the formula or not. For this we use the CTL method, also called model checking a path, by Markey et al. [11]. This means we create a table and check whether the first position in the last row is true. If this is the case, the word model satisfies the formula.
In our program, the table is represented through a map. It maps each part of the formula to a boolean array. This array contains for each position in the word model whether the formula beginning from that position is fulfilled at this position or not. We start with the smallest parts of the formula and use the previous results for computing the results for the other parts of the formula. Finally, we check whether the first position of the array for the whole formula is satisfied and return this as a result. After that we count the results with value $true$ and return the percentage we get from dividing the sum through the number of generated word models.

Counting Word Models via Monte Carlo with Loops Counting word models with loops is very similar to the approach without loops. The only difference is the word model generation. We start with generating the models like we did without loops. Then we need to add the loop to the model. In order to do this, we chose the position l the loop-back transition points to at random. Then we add the part of the word model from l to the end to the word model, we generated first. Finally, we proceed as described above.

5.3.2 Tree Models

For tree models, we implemented three different approaches. Two are bounded model counting approaches, where one of them counts the number of bases and the other one the number of tree models. The third approach is based on the Monte Carlo technique and only counts tree models.

Bounded Model Counting for Tree Models

The implementations of these approaches are also based on the Bounded Model Checking approach known from the work of Clarke et al. [2]. We take both infinite and finite tree models into account and bound them by the depth of the base of the tree model k. As input the program gets the formula, the bound, the number of atomic propositions and the number of trees it should generate.

Bounded Model Counting without Loops First of all, the atomic propositions are generated similar to the generation in the other approaches. In this case, they are represented through strings. Before applying the formulas defined in Definition 12, we parse the string that represents the LTL formula and transform the formula into positive normal form (PNF). This transformation is done by the tool presented in the approach form Giannakopoulou et al. [6]. Note that, for this approach, we only use the translation into PNF and not the translation into Büchi automata.
Next, we generate the desired number of tree models. A tree model is represented as a list of nodes and a node contains information about whether it is a root or a leaf or both as well as information about its child nodes and parent node. Moreover, information about the path of the tree it belongs to and its id are stored in the node.
After that, we enter a loop that generates and checks the formulas for all generated tree models. First, we apply the bounded model checking formulas. Since we are not considering loops, we only need to change the formulas as presented in Definition 12 by adding information about the paths. Next, we convert the tree model into a SAT formula and connect it with the SAT formula we gained out of the LTL formula through an $\wedge$-operator.

The next steps are the CNF conversion and the DIMACS conversion, which are similar to their counterparts for word models.
Since Relsat cannot be embedded into our Java code, we have to write the DIMACS encoding of the formula into a file, before calling it as an external program and redirecting its output back to our Java program. This time, Relsat only checks whether the the formula is satisfiable. We increase our counter if the formula is satisfiable and in the end, the counter is returned.

Bounded Model Counting with Loops The implementation of this approach is almost similar to the previous one. The only difference is that we transform the LTL formula into the SAT formula by applying different rules and that we use Relsat not only for satisfiability checking, but also for counting models. We return the sum of all counted models as our result.

Model Counting via Monte Carlo

As an input the program receives the formula represented through a string, an integer that represents the number of atomic propositions, an integer that represents the depth of the tree and

an integer that denotes the number of trees we want to generate. First, the formula is translated into a Büchi automaton by the tool presented by Giannakopoulou et al. [6]. Next, we generate a set of random trees. The representation of the tree is the same as described in the previous section except that we now explicitly store loop-back transitions in our datastructure.

After that, the product transition system is built. It is represented by the list of initial states of the transition system. These nodes consist of a state of the automaton, a node of the tree the transition system is build on and of a list of successor nodes. Finally, it is checked whether the tree model fulfills the formula by applying the nested DFS and generating transitions on the fly.

CHAPTER 6

EXPERIMENTS AND EVALUATION

6.1 Setup and Benchmarks

We have two different setups for our experiments. We ran the experiments for tree models on an Intel Core i5 processor with 2.3 GHz and 4GB virtual memory. Our approaches for word models are run on an Intel(R) Xeon(R) CPU E5-4650L 0 processor at 2.60GHz. This machine has 786 GB RAM and 64 cores. Our two benchmarks are the performance of the algorithms and the exactness of the results of the approximate counters. The performance is determined by the runtime of the program by steadily increasing the bound. The exactness of the results is determined by comparing the approximate results to the exact ones. Both kinds of results are very important to determine the usability of our approaches. We run ApproxMC with tolerance 0.7 and confidence 0.3.
We test several formulas which are either used in practice [7], are part of the GR1-formulas [13] or introduce an interesting behavior for some of the counting algorithms.

6.2 Word Models

We are considering a Monte Carlo approach and several bounded model counting approaches in this section. First, we consider word bases and determine the quality of the results. Next, we do the same for the approaches that consider word models and finally we take a look at the runtime of all our algorithms.

6.2.1 Word Models without Loops

We begin with a comparison of all the base counting algorithms. The results of this comparison are showed in Figure 6.1. In this figure, we compare our approximate algorithms with the exact algorithm BMC Relsat. The percentage shown in this diagram is computed by dividing the number of all models that fulfill the specification through the number of all models (for the counters based on bounded model checking) or the number of generated models (for the counters based on Monte Carlo). We call the difference between the exact count and the count of the approximate algorithm a deviation. The average deviation is computed by summing up all the deviations of an algorithm and dividing it through the number of formulas.

In general, all approaches perform good. The Monte Carlo approach has an average deviation of 8.36%, its maximum deviation is 20% and its minimal is 3%. For bounded model counting

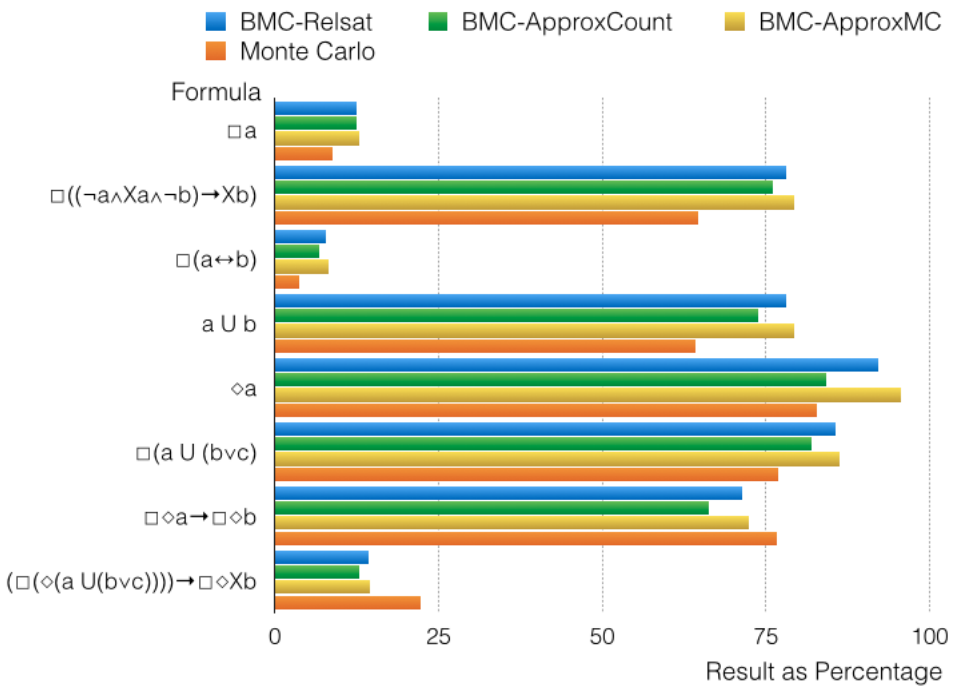


Figure 6.1: Results of the word base counting algorithms with bound 3 and 3 variables

with ApproxCount, the average deviation is 3%, the maximal deviation is 10% and the minimal deviation is 0%. For the ApproxMC approach, the average deviation is 1%, the maximal one is 3% and the minimal one is 0.3%. The ApproxMC approach clearly performs best. It does guess the result almost correctly for each of the tested formulas, although we run it with a relatively high tolerance and a low confidence. Therefore, we can recommend using this approach for both liveness and safety properties. In general, the Monte Carlo Approach yields good results for both checking safety and liveness properties and the ApproxCount algorithm computes very good results considering safety properties. For safety properties like $\Box a$, the ApproxCount approach is even more accurate than the ApproxMC approach.

In contrast, both the ApproxCount and the Monte Carlo approach introduce some outliers. Most outliers for the Monte Carlo approach occur by executing a formula that contains an Until-operator. The outliers of the ApproxCount algorithm occur when checking liveness properties like $\Diamond a$. Especially then the ApproxCount algorithm is very unstable, as we can see in Figure 6.3.

However, the greatest problem of the bounded model counting algorithm, especially the algorithm that uses ApproxMC, is that they do not scale when we are regarding difficult formulas with a higher input. We run the algorithms again, with more variables and a higher bound. The results are shown in Figures 6.2. All algorithms except the Monte Carlo algorithm have no results for several formulas. All except one of these results were not computed, because the CNF conversion did not scale. The result for the formula $\Box\Diamond a \rightarrow \Box\Diamond b$ by running ApproxCount was not computed because ApproxCount terminated after beginning the first iteration. Either it was because the CNF formula was to complicated for the SAT counter or a program error occurred. Since we were executing the formula several times and did not receive a result, we believe that

the former explanation is correct. Note that ApproxMC was not able to compute a single result for this comparison, because it did not terminate even after five days. As you can see later on, ApproxMC is the slowest algorithm and therefore does not scale properly.

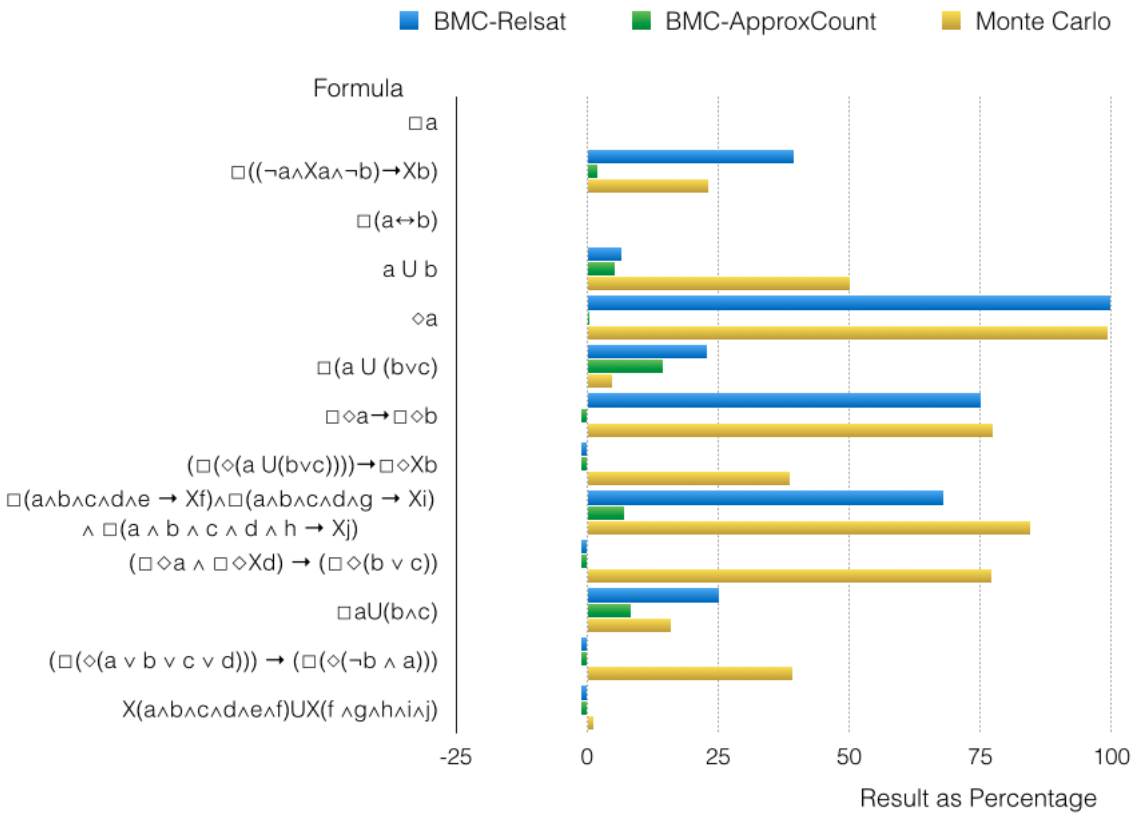
Another important issue, besides the errors, is of course the accuracy of the results from our two approximate approaches for this size of the models. Thereby, the results are very different. It is highly dependent on the formula, which algorithm performs better. On average, the Monte Carlo approach differs from the exact approach about 12%. The highest deviation of this algorithm are about 40% and the smallest is about 0.1%. The average deviation from the approximate bounded model counting algorithm using ApproxCount is 28%. Moreover, its highest deviation is 98%, whereas its smallest deviation is 0%. ApproxCounts high deviation is due to its use of SampleSat [17]. In each iteration, SampleSat draws the samples of our formula. However, it does only perform near uniform sampling and therefore there can be a deviation from uniform sampling. And this deviation leads to such big miscalculations as for the formula $\Diamond a$. In general, the ApproxCount approach performs really good for universality checking and safety properties. It is worse especially for existence checking and liveness properties. If we would not consider formulas belonging to the existence pattern, the average deviation of this approach would be much smaller.

When we run the ApproxCount approach 10 times for the same formulas, we can see that sometimes the average result does not differ much from the result gathered in one run. For some formulas, especially formulas that express liveness properties, this is not the case. One example is $\Diamond a$. As shown in Figure 6.3, the result space differs from 10% of all models fulfilling the formula to 86% of models fulfilling the formula. This high deviation makes the use of the ApproxCount approach highly impractical, at least for formulas that check the existence of a certain specification.

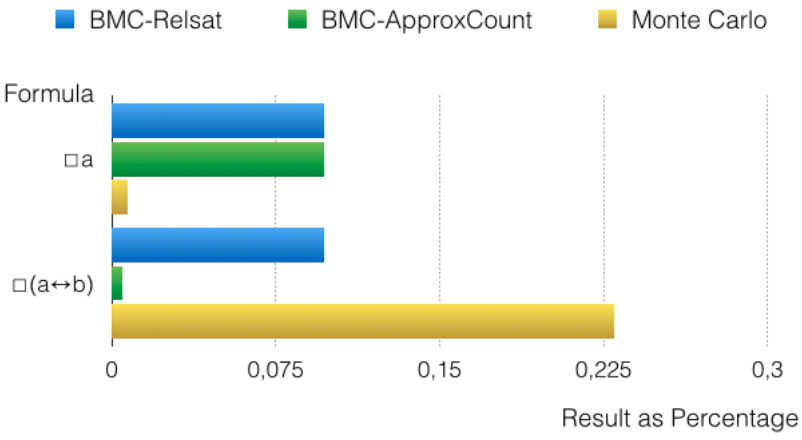
For the approximate bounded model counting approach with ApproxCount, the deviation from the exact solution increases for the average value of ten calls. However, there is still a big difference between the best guess the algorithm makes and the worst one and that difference almost does not change when we run the algorithm several times and compute the average value. For some formulas, the average value is closer to the exact value. Nevertheless running the algorithm several times decreases the highest deviation. This is the highest benefit of multiple runs against the single run. We think, that, in this case, multiple runs do not pay off. They cost much more time than a single run and although the results are better than the guesses of the single run, the difference is not that high.

For the Monte Carlo approach, the average, lowest and highest deviation from the exact solution is the same for the ten calls as for just one call. In general, the average results also do not differ very much from the single ones. The ten single results do only differ in the decimal places, if they differ at all. So basically, by using this approach, we obtain almost the same results for a formula independent of the number of runs.

In general, with small numbers, all our algorithms perform good in counting, but ApproxMC is the best. Its only deficit from the experiments we run, is its performance as also shown in one of the following sections. For this reason, it is best to use our exact approach. It guarantees the exact results and it outperforms all our approximate approaches, as shown in a section



(a) Part 1



(b) Part 2

Figure 6.2: Accuracy of the algorithms for bound 10 and 10 variables

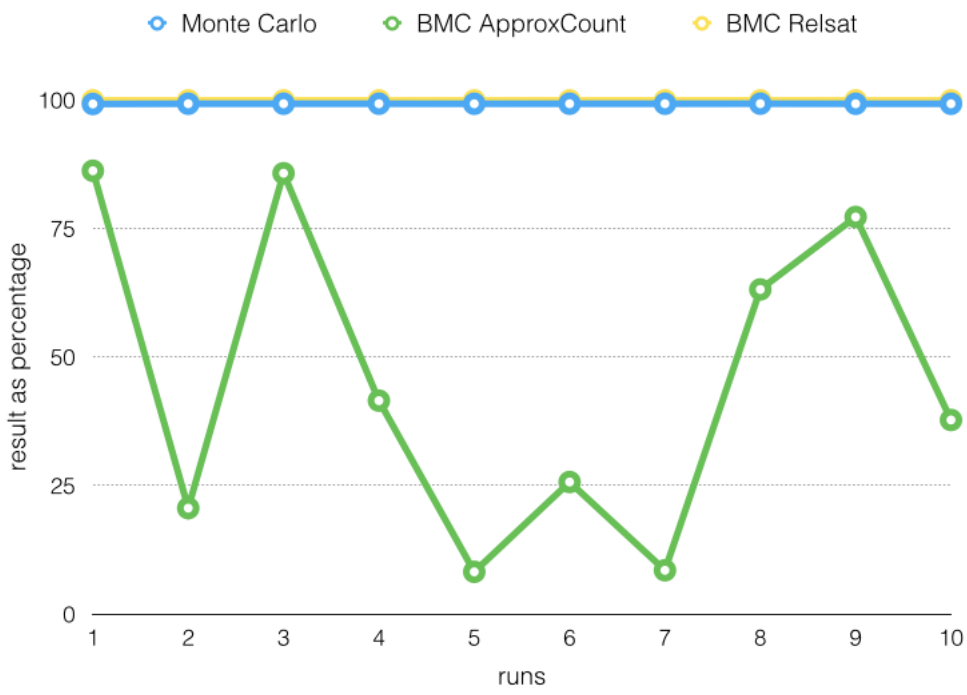


Figure 6.3: Results of ten runs of ApproxCount with bound 10 and 10 variables for the formula $\Diamond a$

below. For approximate counting of models of bound three, both the Monte Carlo approach and the ApproxCount approach do a good job. It is really difficult to determine which of them, we should choose. For safety properties, the approximate bounded model counting approach with ApproxCount yields the better results than the Monte Carlo approach and we recommend using the ApproxCount approach when we check formulas belonging to universality patterns. Therefore, it delivers almost exact results. But when checking liveness properties, it performs worse than the Monte Carlo approach. So, if you want an algorithm that guesses the right count with an deviation not greater than 20%, the Monte Carlo approach is the better one. If you want highly exact results for safety properties and you do not check liveness properties, then you should choose the approximate bounded model counting approach with ApproxCount. When considering models of bound 10, it is better to choose the Monte Carlo approach. The Monte Carlo approach is able to compute results for all tested formulas and has a better average deviation than the ApproxCount approach. Furthermore, running the approximate algorithms multiple times does not pay off and leads to almost no difference in the results.

6.2.2 Word Models with Loops

In this section, we focus on the accuracy of our approaches that tackle LTL counting for word models with loops. In Figure 6.4, the results are shown. Note that -1 as a result means that the algorithm did not scale for this formula. The reasons for this failures are explained below.

The average deviation of the ApproxMC approach is 0.8%, its maximal deviation is 2% and its minimal deviation is 0.01%. For the ApproxCount approach, the average deviation is 17%, its maximal deviation is 40% and its minimal one is 0.6%. Moreover, the Monte Carlo approach

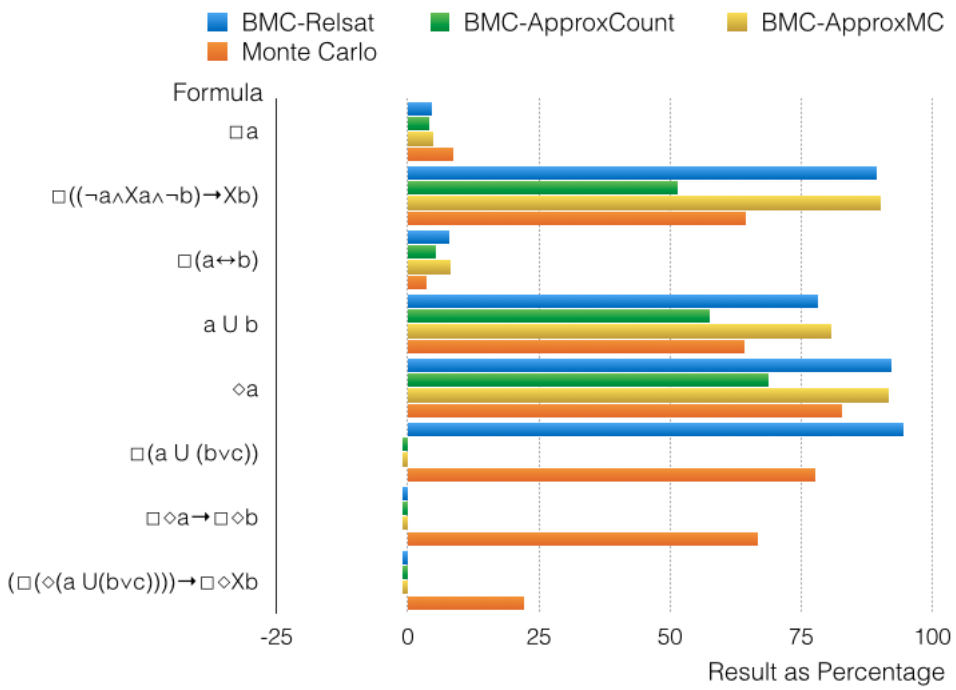


Figure 6.4: Results of the word model counting algorithms with bound 3 and 3 variables

has an average deviation of 12%, a maximal deviation of 25% and a minimal deviation of 4%.

Again, the ApproxMC approach computes the best results and again it has problems with the performance. The ApproxCount approach performs worst considering the accuracy. The formulas ApproxCount fails the most are $\Diamond a$, aUb and $\Box((\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$. After getting the results of ApproxCounts high deviations regarding the formula $\Diamond a$ in the previous section, this outcome does not surprise us. Again, we can recommend using the ApproxCount approach for checking safety properties. For the Monte Carlo approach, there are plenty of deviations of different degrees. It has low deviations for formulas that only have a few solutions. Its worst deviations are for $\Box((\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$ and aUb However, if we want to use an approach that scales good and yields good results for both liveness and safety properties, then we should use the Monte Carlo approach.

Basically, all bounded model counting approaches can deliver good results and can be used without problems, except that they all suffer from performance problems. Most of the errors in Figure 6.4 result out of the failure of the CNF conversion. However, the error for the formula $\Box(aU(b \vee c))$ does not. As shown, the Relsat approach computes a result for this formula, but the ApproxMC and the ApproxCount approach do not. They fail, because of the SAT counter instance. We do not know exactly why they failed, but we think that they both had a timeout. However, we cannot determine the reason for this failure for sure.

6.2.3 Performance experiments

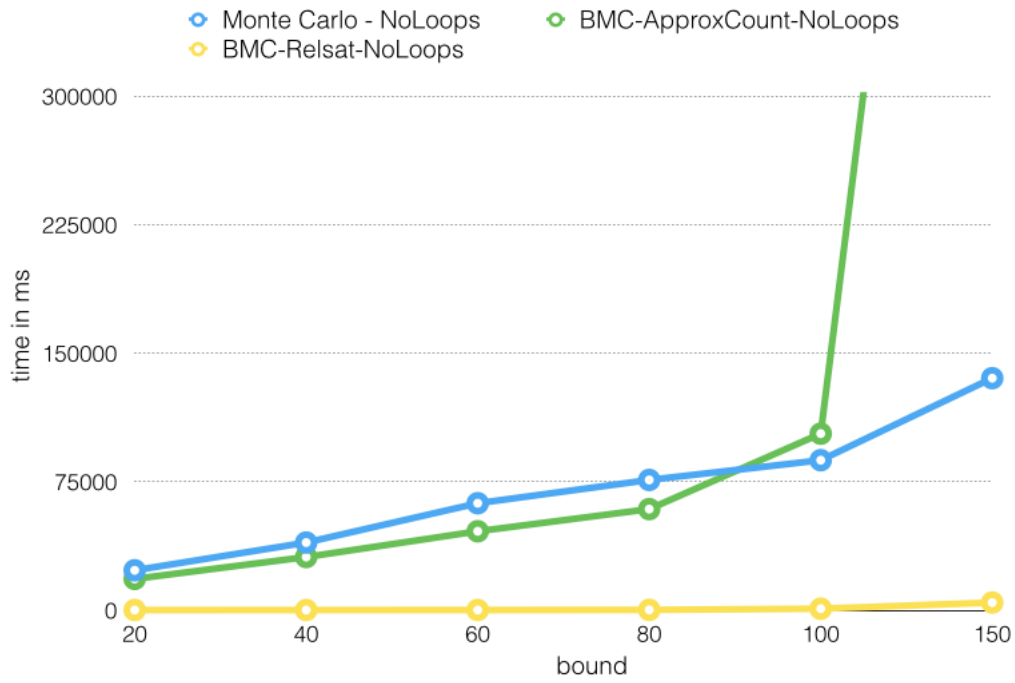
In order to get an estimate about the performance of our programs, we increased the bound of the models continuously. We ran these experiments for both the formulas $\Diamond a$ and $\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$ and 10 variables and created 1 million models for the Monte Carlo approach. Since the bounded model counting approaches for loops do not scale, we first take a look at the approaches for the bases.

The results are shown in Figure 6.5 and 6.6. The Relsat counting approach clearly performs best in both cases. It performs almost more than 20 times better than the other approaches although it counts exact. However, we believe that this good performance is not possible for all kinds of formulas. For example, all approaches based on bounded model counting do not scale for the GR1 formulas with bound 10 and 10 variables. Sometimes, the CNF conversion is responsible for that, but not in every case.
At first, for the formula $\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$, the Monte Carlo approach and the ApproxCount approach perform almost equivalent. Note that we can make the performance of the Monte Carlo algorithm better by generating less models. The accuracy performance between 100000 and 1000000 generated models does not differ much. However, after considering models with a bound higher than 20, the runtime of the Monte Carlo approach increases faster than the one of ApproxCount. This changes when we are considering a bound of 90. Then they need exactly the same amount of time to compute the models. After that the Monte Carlo approach is faster and for bound 150 the ApproxCount approach does not scale anymore.

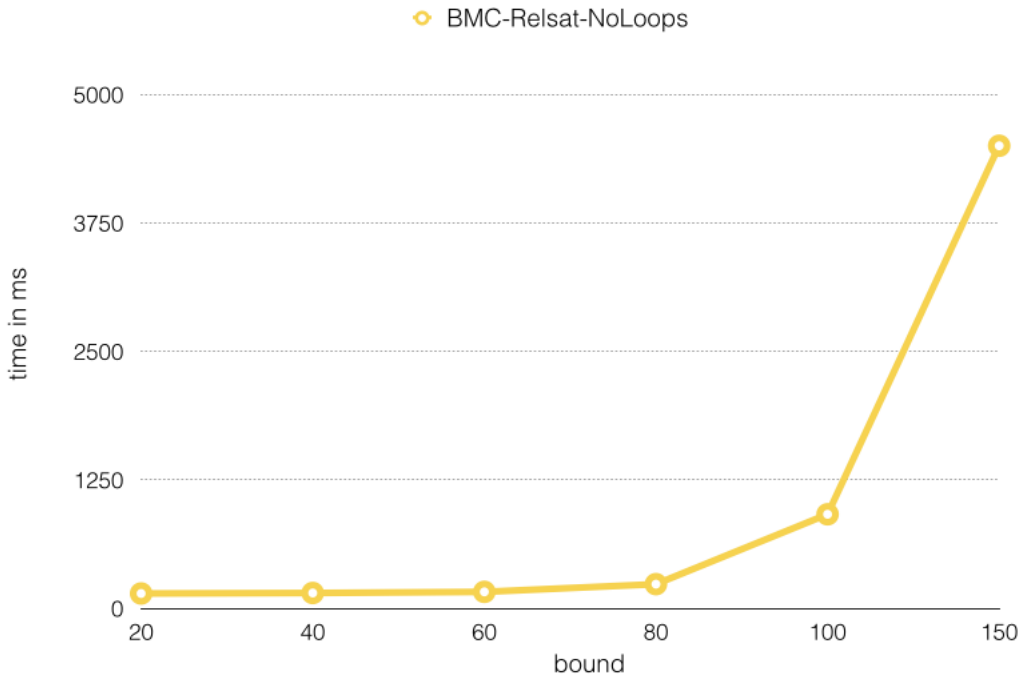
The results for the formula $\Diamond a$ slightly differ from the one for the formula $\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$. Again, Relsat is the fastest approach and the ApproxCount approach does not scale anymore for bound 150. For bounds lower than 150, the ApproxCount approach shows a much better performance than before, its runtime decreases from bound 20 to bound 60 and starts increasing again when it starts considering a bound of 80. The reason for this strange behavior is again the unstableness of ApproxCount for this kind of formula. By running ApproxCount 10 times with this formula, we could observe the unstableness in the results as well as the unstableness in the runtime. The worse the results for accuracy are, the better the runtime gets. This triggers the decreasing of the runtime for the runs from bound 20 to 60.

Note that there is a reason why no runtime of the ApproxMC approach is shown in any of the figures. The reason is that after five days of running the approach for the formulas and bound 20, there was no result computed. This approach is very slow and responsible is the SAT counter ApproxMC. First, we wanted to run the algorithm with a relatively low tolerance and high confidence, but then no result was received. Later on, we decided to run it with high tolerance and low confidence and still no result was computed. Therefore this algorithm is highly impractical for any practical use although its results in the accuracy tests were quite good.

For word models with loops, all bounded model counting approaches did not scale. The primary reason for this is the CNF conversion that cannot handle formulas that large. By now, we did not found a CNF converter that solves this problem and we cannot transform the bounded model checking formulas such that the formula is already in CNF. In contrast, the Monte Carlo ap-



(a) Performance for all Algorithms



(b) Scaling for the Relsat approach

Figure 6.5: Performance of the algorithms for word models for the formula $\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$ considering 10 variables

proach performs nearly as good with loops as it does without and is on average 9000 ms slower than its counterpart without loops for the formula $\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$. For the formula $\Diamond a$, the difference is even smaller. On average, the approach is only 500 ms slower than its counterpart without loops.

We also compared the performance of both our exact algorithms. The results are shown in Figure 6.7.
Again, the Relsat approach counting word bases is the fastest and the Word Counting algorithm presented in [5] is the second best. The Relsat approach for word models did not scale for any of our inputs. The reason for this is again the CNF conversion.

Of course, one might say that the comparison of the Relsat approach counting bases with the Word Counting algorithm might not be fair and clearly it is not, but it gives us an idea how the Relsat approach for loops would perform if we would not have the CNF conversion problem. We think that for certain formulas it could be faster than the Word Counting algorithm and moreover, it would also be possible to determine the count also for formulas that do not specify safety properties. However, for now the Word Counting algorithm presented in [5] remains the best exact algorithm for the word model counting problem.
Finally, we compare the performance of the algorithm presented in [5] to the performance of our approximate algorithms for loops. Since only our Monte Carlo approach scales, we compare it only to that approach. We run the algorithms considering 10 atomic propositions and check the formula $\Box a$. The Monte Carlo algorithm generates 100000 models. The results are shown in Figure 6.8.
The approximate algorithm clearly performs better, but the runtime of the exact algorithm is not bad either. Note that when decreasing or increasing the number of models the Monte Carlo approach generates, the runtime can differ.

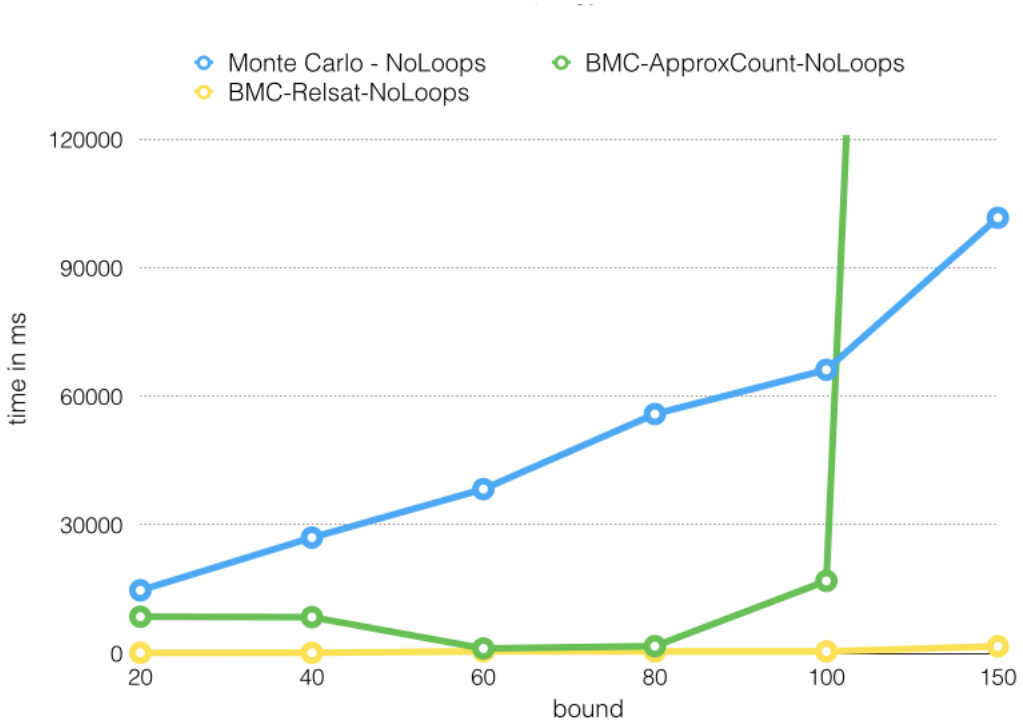
6.3 Tree Models

We introduced three approximate counting approaches for tree models. One of them considers tree models that do not have any loops and two of them focus on tree models with loops. Moreover, one of the approaches is based on automata based model checking and the other two are built on bounded model checking. First, we take a look at the correctness of the approach for tree bases. Next, we consider the correctness of the two approaches for tree models and finally we compare their performance.

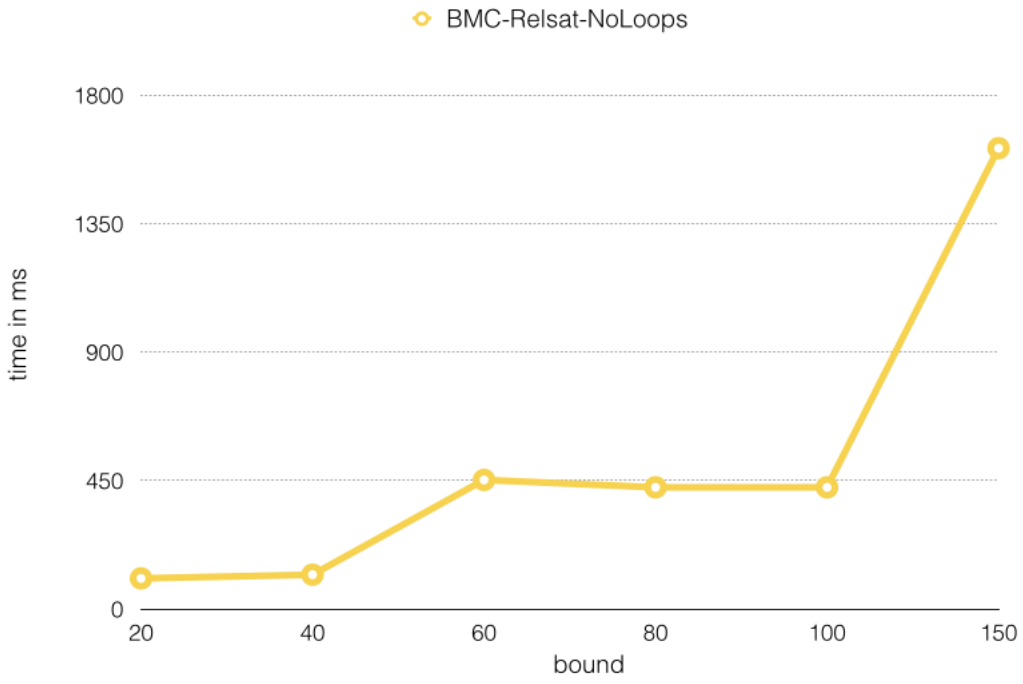
6.3.1 Counting Tree Models without Loops

In Figure 6.9, the results of the bounded tree model counting approach without loops and the exact results are compared.

The average deviation of the approximate count is 4.25%. This is a really low deviation. The maximal deviation is 7% and the minimal deviation is 0.5%. Both the best and the worst guess of this algorithm are obtained when taking a GR1-formula into account. Since this approach is faster than our brute-force exact counting algorithm and derives almost as good results, it is



(a) Performance for all Algorithms



(b) Scaling for the Relsat approach

Figure 6.6: Performance of the algorithms for word models for the formula $\Diamond a$ considering 10 variables

highly recommended to use it for both safety and liveness properties. Although, it is based on bounded model checking, it can also provide good guesses for trees with a greater depth.

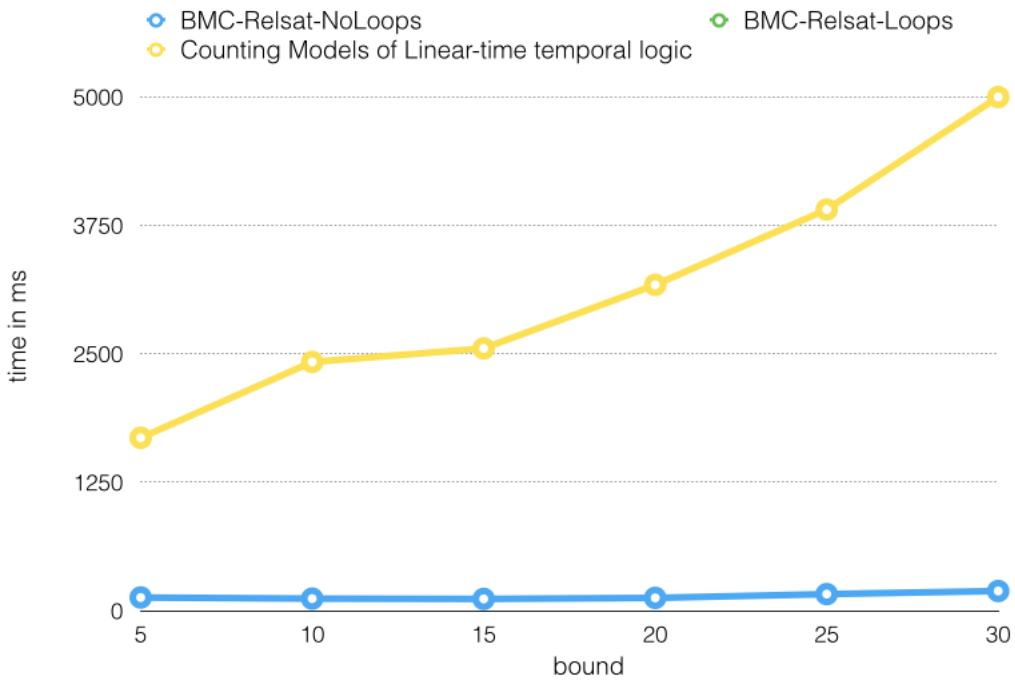


Figure 6.7: Performance of the exact algorithms for the formula $\Box a$ considering 10 variables

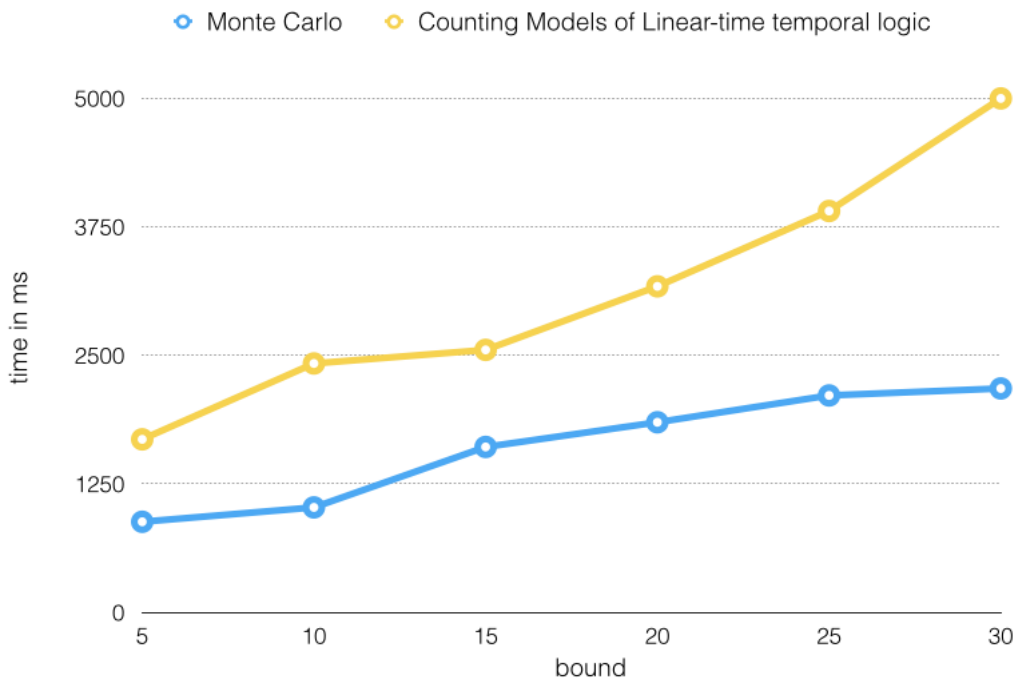


Figure 6.8: Performance of the exact algorithm and the approximate algorithm for the formula $\Box a$ considering 10 variables (loops)

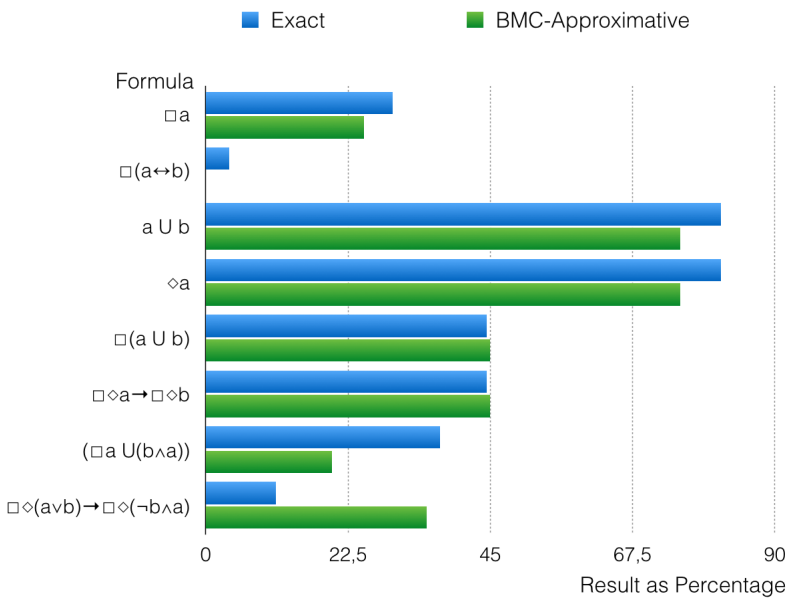


Figure 6.9: Results of the tree base counting algorithms with depth 2 and 2 variables and 20 generated models

6.3.2 Counting Tree Models with Loops

In Figure 6.10, we show the results of the experiments for tree models with loops. We compared the bounded model counting approach and the automata based model counting approach with the exact results.

As shown, the results are quite good. The average deviation of the bounded model counting approach is 4%, its maximal deviation is 9% and its minimal deviation is 1%. For the automata based algorithm, the average deviation is 5%, the maximum deviation is 15% and the minimal deviation is 0.3%. In general, both approaches perform good. The only large deviations for the automata based approach occur by executing the formulas $\Box(aUb)$ and $\Box a$. Therefore, for universality checks and for checking safety properties choosing the bounded model counting approach could be better. The largest deviations for the bounded model counting approach occurs when executing the formulas $\Diamond a$ and aUb and therefore, existence checking and checking liveness properties is better performed by the automata based approach.

If we want a good and fast estimate where we do not need to be afraid that is does not scale, we choose the automata based approach. The results of the counting are quite promising and in the few cases, where bounded model counting is better, the difference is really small. Moreover, we can recommend using this approach for both liveness and safety properties.

6.3.3 Performance Experiments

We run our algorithms for tree models for the formula $\Diamond a$ and increase the bound from run to run. We start with bound 5 and increase it until bound 20. We consider two variables for the trees and generate 1000 trees in each run. The results are shown in Figure 6.11.
The automata based tree counting approach clearly performs best. The BMC tree counting

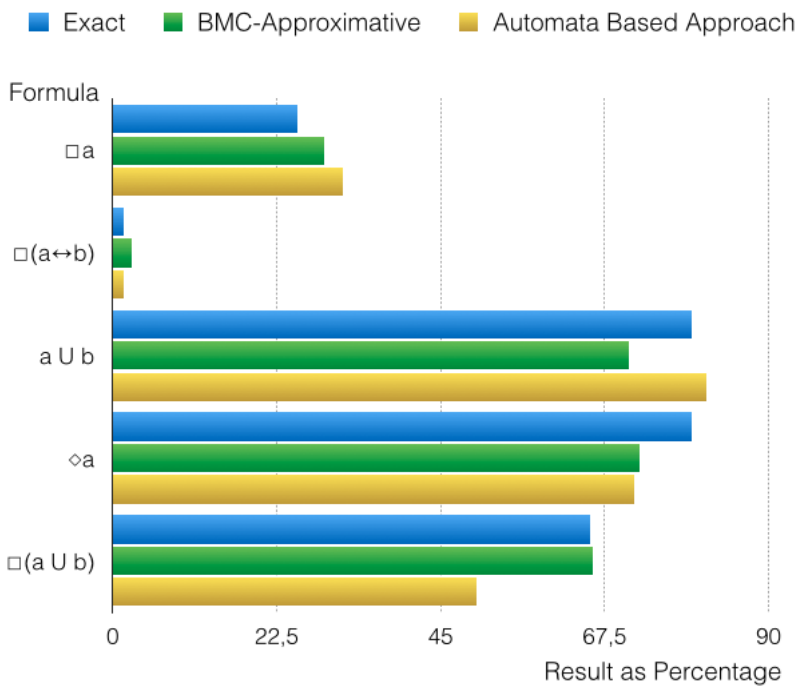


Figure 6.10: Results of the tree model counting algorithms with depth 2 and 2 variables and 20 generated models

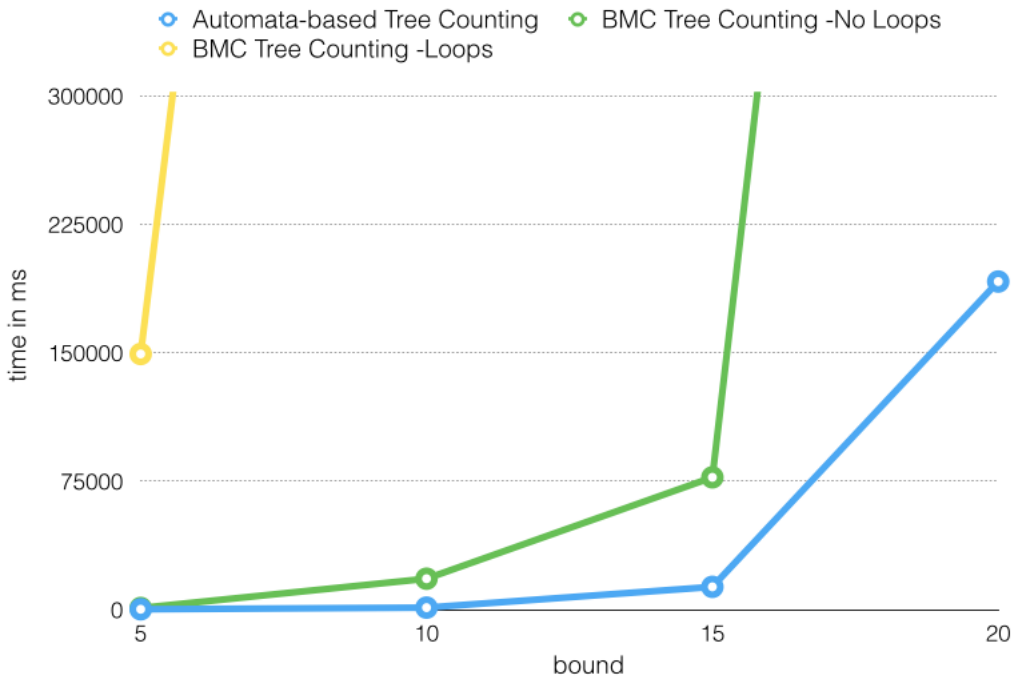


Figure 6.11: Performance of the algorithms for trees for the formula $\Diamond a$ considering two variables

approach for bases does not scale anymore when the bound is higher than 15 and the BMC tree counting approach for loops does not scale anymore when the bound is higher than 5. In contrast, the automata based approach can count trees with a depth greater than 25 without any problems.

However, converting the SAT formula into CNF is again responsible for this result. The algorithms based on bounded model counting need to convert 1000 formulas, one for each generated tree, into CNF. Especially by converting LTL formulas that consider loops or considering a high bound, the SAT formulas get very large, such that converting them to CNF results in an exponentially large formula which then is unfeasible to compute. A solution to that problem is finding a faster way of converting the formula into CNF, although as far as we know, our CNF converter is state of the art.

Note that the comparison of all this algorithms might not be totally fair. First, not all algorithms consider models with loops. Second, the automata based approach builds tree models and then computes whether the formula is fulfilled whereas the BMC tree counter for loops generates the base of the models and then applies the bounded model counting formulas for models which contain loops. Therefore, the bounded model counter can count much more models in a run that generates ten bases than the automata based approach can when it generates ten models. However, the performance of the automata based approach remains best.

In general, the bounded tree model counting algorithm for tree bases should be fine for almost all formulas and depths that are not greater than fifteen. By counting tree models, we would recommend using the automata based algorithm because of its high performance benefits.

CHAPTER 7
CONCLUSION

We presented a few LTL counting algorithms for tree and word models. For word models, we presented one exact and two approximate model counting algorithms as well as an algorithm based on the Monte Carlo method for both counting the bases of the word models and for counting the models themselves. For tree models, we presented one approximate bounded model counting algorithm that tackles both the counting problem for tree bases as well as the problem for tree models and an approximate automata based approach based on Monte Carlo that is only capable of counting tree models.

We evaluated our algorithms on certain formulas and compared their performance as well as the accuracy of the approximative algorithms.
Concerning the accuracy of the algorithms, all algorithms for word models and tree models perform good. For word models, all algorithms except the ApproxMC algorithm show weaknesses for certain kinds of formulas. Thereby, the weakness of the ApproxCount approach includes mostly formulas for existence checking and for liveness properties, whereas for the Monte Carlo approach, its weakness are formulas that contain the Until-operator. Considering tree models, no big outliers were found and basically both approaches can be used for checking both liveness and safety properties.

Concerning the performance, all algorithms based on bounded model counting failed. It was not possible to compute results for high bounds and lots of variables because of the high complexity of transforming a SAT formula into CNF. When considering models that contain loops, it also was not impossible to compute results of GR1-formulas for small bounds. Therefore, the bounded model counting approaches are impractical in practice until the CNF conversion complexity problem is solved. The other approaches performed quite well and we can highly recommend using the Monte Carlo approach for word models or the automata based approach for tree models instead of an exact counting algorithm.

For future work, it would be interesting to run these algorithms on more formulas with a higher bound and more variables. Due to the limited number of formulas we could compute because of scaling problems with the exact algorithms and the limitation of the algorithm presented in [5] to safety properties, this would be useful to get even more significant results. Furthermore, it would also be interesting to develop a LTL counting approach that does not rely on certain kinds of models like word or tree models.

CHAPTER 8

APPENDIX

Here we present useful information, that is not necessarily needed to understand the thesis.

8.1 Experiments

8.1.1 Word Models

Word Models without Loops

We run our experiments for our Monte Carlo approach, and the bounded model counting approaches. You can see the results for the first one in Table 8.4, the results for the exact bounded model counting approach in Table 8.3 and the results for the approximate bounded model counting approaches in Table 8.1 and 8.2.

In Tables 8.5, 8.6 and 8.7, the results for the Monte Carlo approach, the Relsat Approach and the ApproxCount approach for bound 10 and 10 variables are shown.

Formula	% of satisfying Word Models
$\Box a$	12.89
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	79.3
$\Box(a \leftrightarrow b)$	8.16
aUb	79.3
$\Diamond a$	95.62
$\Box(aU(b \vee c))$	86.29
$\Box\Diamond a \rightarrow \Box\Diamond b$	72.3
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	14.75

Table 8.1: Results of approximate Counting via Bounded Model Counting and ApproxMC [4] with bound 3 and 3 variables (no loops)

Formula	% of satisfying Word Models
$\Box a$	12.5
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	76.09
$\Box(a \leftrightarrow b)$	6.7
aUb	73.76
$\Diamond a$	84.25
$\Box(aU(b \vee c))$	81.92
$\Box\Diamond a \rightarrow \Box\Diamond b$	66.18
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	12.82

Table 8.2: Results of approximate Counting via Bounded Model Counting and ApproxCount [18] with bound 3 and 3 variables (no loops)

Formula	% of satisfying Word Models
$\Box a$	12.5
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	78.13
$\Box(a \leftrightarrow b)$	7.87
aUb	78.13
$\Diamond a$	92.12
$\Box(aU(b \vee c))$	85.71
$\Box\Diamond a \rightarrow \Box\Diamond b$	71.42
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	14.28

Table 8.3: Results of exact Counting via Bounded Model Counting and Relsat [1] with bound 3 and 3 variables (no loops)

Formula	% of satisfying Word Models
$\Box a$	8.7
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	64.56
$\Box(a \leftrightarrow b)$	3.7
aUb	64.19
$\Diamond a$	82.86
$\Box(aU(b \vee c))$	77.78
$\Box\Diamond a \rightarrow \Box\Diamond b$	76.72
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	22.22

Table 8.4: Results of approximate Counting via Monte Carlo with bound 3 and 3 variables and 10000 generated models (no loops)

Formula	% of satisfying Word Models
$\Box a$	0.0074
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	23.09
$\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$	84.43
$\Box(a \leftrightarrow b)$	0.23
aUb	50.14
$\Diamond a$	99.2
$\Box(aU(b \vee c))$	4.66
$\Box\Diamond a \rightarrow \Box\Diamond b$	77.33
$(\Box\Diamond a \wedge \Box\Diamond Xd) \rightarrow (\Box\Diamond(b \vee c))$	77.17
$\Box aU(b \wedge c)$	15.92
$(\Box(\Diamond(a \vee b \vee c \vee d))) \rightarrow (\Box(\Diamond(\neg b \wedge a)))$	39.28
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	38.55
$X(a \wedge b \wedge c \wedge d \wedge e \wedge f)UX(f \wedge g \wedge h \wedge i \wedge j)$	1.13

Table 8.5: Results of Model Counting Algorithm via Monte Carlo without Loops and with 10 as bound, 10 as the size of the set of AP and 1000000 generated models

Formula	% of satisfying Word Models
$\Box a$	0.097
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	39.52
$\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$	67.86
$\Box(a \leftrightarrow b)$	0.097
aUb	6.67
$\Diamond a$	99.9
$\Box(aU(b \vee c))$	22.77
$\Box\Diamond a \rightarrow \Box\Diamond b$	74.97
$(\Box\Diamond a \wedge \Box\Diamond Xd) \rightarrow (\Box\Diamond(b \vee c))$	no result
$\Box aU(b \wedge c)$	25.09
$(\Box(\Diamond(a \vee b \vee c \vee d))) \rightarrow (\Box(\Diamond(\neg b \wedge a)))$	no result
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	no result
$X(a \wedge b \wedge c \wedge d \wedge e \wedge f)UX(f \wedge g \wedge h \wedge i \wedge j)$	no result

Table 8.6: Results of exact Counting via Bounded Model Counting and Relsat [1] with bound 10 and 10 variables as input (no loops)

Formula	% of satisfying Word Models
$\Box a$	0.097
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	1.84
$\Box(a \wedge b \wedge c \wedge d \wedge e \rightarrow Xf) \wedge \Box(a \wedge b \wedge c \wedge d \wedge g \rightarrow Xi) \wedge \Box(a \wedge b \wedge c \wedge d \wedge h \rightarrow Xj)$	7.01
$\Box(a \leftrightarrow b)$	0.005
aUb	5.34
$\Diamond a$	0.46
$\Box(aU(b \vee c))$	14.54
$\Box\Diamond a \rightarrow \Box\Diamond b$	no result
$(\Box\Diamond a \wedge \Box\Diamond Xd) \rightarrow (\Box\Diamond(b \vee c))$	no result
$\Box aU(b \wedge c)$	8.46
$(\Box(\Diamond(a \vee b \vee c \vee d))) \rightarrow (\Box(\Diamond(\neg b \wedge a)))$	no result
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	no result
$X(a \wedge b \wedge c \wedge d \wedge e \wedge f)UX(f \wedge g \wedge h \wedge i \wedge j)$	no result

Table 8.7: Results of approximate Counting via Bounded Model Counting and ApproxCount [18] with bound 10 and 10 variables as input (no loops)

Word Models wih Loops

In Tables 8.8 and 8.9, the results for the ApproxMC and ApproxCount approach are showed. Moreover, the results for the Monte Carlo approach and the Relsat approach are shown in Tables 8.11 and 8.10.

Formula	% of satisfying Word Models
$\Box a$	4.78
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	90.18
$\Box(a \leftrightarrow b)$	8.13
aUb	80.85
$\Diamond a$	91.73
$\Box(aU(b \vee c))$	no result
$\Box\Diamond a \rightarrow \Box\Diamond b$	no result
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	no result

Table 8.8: Results of approximate Counting via Bounded Model Counting and ApproxMC [4] with bound 3 and 3 variables (loops)

Formula	% of satisfying Word Models
$\Box a$	4.02
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	51.5
$\Box(a \leftrightarrow b)$	5.24
aUb	57.53
$\Diamond a$	68.8
$\Box(aU(b \vee c))$	no result
$\Box\Diamond a \rightarrow \Box\Diamond b$	no result
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	no result

Table 8.9: Results of approximate Counting via Bounded Model Counting and ApproxCount [18] with bound 3 and 3 variables (loops)

Formula	% of satisfying Word Models
$\Box a$	4.68
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	89.31
$\Box(a \leftrightarrow b)$	7.87
aUb	78.13
$\Diamond a$	92.12
$\Box(aU(b \vee c))$	94.46
$\Box\Diamond a \rightarrow \Box\Diamond b$	no result
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	no result

Table 8.10: Results of exact Counting via Bounded Model Counting and Relsat [1] with bound 3 and 3 variables (loops)

Formula	% of satisfying Word Models
$\Box a$	8.8
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow Xb)$	64.63
$\Box(a \leftrightarrow b)$	3.7
aUb	64.22
$\Diamond a$	82.89
$\Box(aU(b \vee c))$	77.73
$\Box\Diamond a \rightarrow \Box\Diamond b$	66,72
$(\Box(\Diamond(aU(b \vee c)))) \rightarrow \Box\Diamond Xb$	22.23

Table 8.11: Results of approximate Counting via Monte Carlo with bound 3 and 3 variables and 10000 generated models (loops)

Formula	% of satisfying Word Models
$\Box a$	25
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow b)$	100
$\Box(a \leftrightarrow b)$	0
aUb	75
$\Diamond a$	75
$\Box(aUb)$	45
$\Box\Diamond a \rightarrow \Box\Diamond b$	45
$\Box aU(b \wedge a)$	30
$(\Box(\Diamond(a \vee b))) \rightarrow (\Box(\Diamond(\neg b \wedge a)))$	20

Table 8.12: Results of approximate Counting via Bounded Model Counting for Trees (no loops)

8.1.2 Tree Models

Tree Models without Loops

Here, the concrete results for the exactness checks of the tree algorithms that do not consider loops are displayed. In Table 8.12, the approximate results are shown and in Table 8.13 the exact reults are shown.

Formula	% of satisfying Word Models
$\Box a$	29.62
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow b)$	100
$\Box(a \leftrightarrow b)$	3,7
aUb	81.48
$\Diamond a$	81.48
$\Box(aUb)$	44.44
$\Box\Diamond a \rightarrow \Box\Diamond b$	44.44
$\Box aU(b \wedge a)$	37,03
$(\Box(\Diamond(a \vee b))) \rightarrow (\Box(\Diamond(\neg b \wedge a)))$	11.11

Table 8.13: Results of exact Counting for Trees (no loops)

Formula	% of satisfying Word Models
$\Box a$	29
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow b)$	100
$\Box(a \leftrightarrow b)$	2.57
aUb	70.88
$\Diamond a$	72.33
$\Box(aUb)$	65.94

Table 8.14: Results of approximate Counting via Bounded Model Counting for Trees (loops)

Counting Tree Models with Loops

In this subsection, we show the concrete results of the exactness checks provided in the experiments chapter. In Table 8.14 and 8.15 the approximate results of counting via bounded model checking and automata-based model checking are shown, whereas in Table 8.16 the exact counts are displayed.

Formula	% of satisfying Word Models
$\Box a$	31.5
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow b)$	100
$\Box(a \leftrightarrow b)$	1.5
aUb	81.5
$\Diamond a$	71.5
$\Box(aUb)$	50
$\Box\Diamond a \rightarrow \Box\Diamond b$	46.5

Table 8.15: Results of approximate Counting via Automata Based Counting for Trees

Formula	% of satisfying Word Models
$\Box a$	25.39
$\Box(\neg a \wedge Xa \wedge \neg b) \rightarrow b)$	100
$\Box(a \leftrightarrow b)$	1.58
aUb	79.36
$\Diamond a$	79.36
$\Box(aUb)$	65.65
$\Box\Diamond a \rightarrow \Box\Diamond b$	-1

Table 8.16: Results of exact Counting for Trees (loops)

Bibliography

[1] Roberto J Bayardo Jr and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208, 1997.

[2] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[3] Julius Richard Büchi. *On a decision method in restricted second order arithmetic*. na, 1960.

[4] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming*, pages 200–216. Springer, 2013.

[5] Bernd Finkbeiner and Hazem Torfah. Counting models of linear-time temporal logic. In *Language and Automata Theory and Applications*, pages 360–371. Springer, 2014.

[6] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of ltl formulae to büchi automata. In *Formal Techniques for Networked and Distributed Sytems—FORTE 2002*, pages 308–326. Springer, 2002.

[7] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A Henzinger. Synthesis of amba ahb from formal specification: a case study. *International Journal on Software Tools for Technology Transfer*, 15(5-6):585–601, 2013.

[8] Carla P Gomes, Ashish Sabharwal, and Bart Selman. Model counting. 2008.

[9] Richard M Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of algorithms*, 10(3):429–448, 1989.

[10] Michael L Littman, Stephen M Majercik, and Toniann Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.

[11] Nicolas Markey and Philippe Schnoebelen. Model checking a path. In *CONCUR 2003-concurrency theory*, pages 251–265. Springer, 2003.

[12] Daniel Morwood and Daniel Bryce. Evaluating temporal plans in incomplete domains. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[13] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[14] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[15] Reuven Y Rubinstein and Dirk P Kroese. Counting via monte carlo. *Simulation and the Monte Carlo Method, Second Edition*, pages 279–314.

[16] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*, pages 322–331. IEEE Computer Society, 1986.

[17] Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, volume 4, pages 670–676, 2004.

[18] Wei Wei and Bart Selman. A new approach to model counting. In *Theory and Applications of Satisfiability Testing*, pages 324–339. Springer, 2005.