Operations on a Symbolic Domain for Synthesis

Felix Matthias Stutz

April 5th, 2017

Bachelor Thesis

University of Kaiserslautern

Department of Computer Science

First reviewer	Prof. Dr. Klaus Schneider
Second reviewer	Prof. Dr. Roland Meyer
Supervisor	M. Sc. Sebastian Muskalla

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kaiserslautern, den 05. April 2017

Felix M. Stutz

Zusammenfassung

Procedure Summaries berechnen den Gesamteffekt von Methoden, sodass nicht jeder Aufruf durch deren Code ersetzt werden muss. Deshalb werden sie häufig bei der Verifikation rekursiver Programme eingesetzt. In [HMM16] wurde für Syntheseprobleme gezeigt, dass *Procedure Summaries* als positive Boole'sche Formeln über dem Transitionsmonoiden eines nichtdeterministischen, endlichen Automaten formuliert werden können. Synthese dreht sich um die Frage, ob man ein lückenhaftes Programm so vervollständigen kann, dass es eine reguläre Spezifikation erfüllt. Dieses Problem kann man durch ein Spiel modellieren, in dem ein Spieler probiert, die Spezifikation zu erfüllen, und der andere durch Veränderungen der Umgebung versucht, dass der erste Spieler nicht gewinnen kann. In [HMM16] wurde eine Methode vorgestellt, mit der solche Spiele gelöst werden können. Mithilfe einer Kleene-Iteration berechnen wir hierfür einen Fixpunkt über diesen positiven Boole'schen Formeln. Um die logische Äquivalenz zweier aufeinanderfolgender Formeln zu erkennen, benötigen wir Implikationstests. Wir zeigen, dass diese Tests co-NP-vollständig sind und präsentieren einen korrekten und vollständigen Sequenzenkalkül, mit dem wir diese symbolisch handhaben können. Wir stellen einen weiteren Ansatz vor, bei dem mithilfe der Monotonie positiver Boole'scher Formeln alle Kandidaten für ein Gegenbeispiel, das die Implikation widerlegen könnte, gesucht und getestet werden. Diesen Ansatz erweitern wir mithilfe eines Verfeinerungsverfahrens, um auch symbolische Formeln handhaben zu können. Des Weiteren erklären wir unser Tool, mit dem wir verschiedene Experimente durchgeführt haben. Mithilfe dieser vergleichen wir die verschiedenen Ansätze für Implikationstests nichtsymbolischer Formeln, indem wir diese Techniken in eine Fixpunktiteration einbetten. Außerdem untersuchen wir den Nutzen verschiedener Vorverarbeitungsmethoden, die wir im Verlauf der Arbeit vorstellen.

Abstract

Procedure summaries are frequently used for verification of recursive programs as they compute the effect of a method instead of inlining its code. In [HMM16], it was shown that positive Boolean formulas over the transition monoid of a non-deterministic finite automaton correspond to procedure summaries for synthesis problems in which one wants to find a completion for a partial program so that all executions satisfy a regular specification. This scenario can be modeled as a two player game in which one player tries to meet the specification whereas the other player tries to adjust the environment so that the first one fails to find a strategy to win the game. In [HMM16], a procedure to solve these games has been presented. By Kleene iteration, we find a fixed-point over those formulas for which we use implication checks to decide whether two succeeding formulas in the iteration are logically equivalent. We prove these checks to be co-NP-complete and propose a sequent calculus to handle them symbolically and prove it to be sound and complete. Moreover, we present an approach in which we exploit the monotonicity of the domain of positive Boolean formulas in order to find and check candidates for counterexamples that disprove the implication. Extending this approach with a refinement procedure enhances it to handle symbolic formulas as well. We explain our tool with which we ran experiments in order to compare the performance of the different algorithms for non-symbolic formulas by embedding these techniques in a fixed-point iteration. Furthermore, we analyze the benefits of different preprocessing procedures which are presented in this thesis.

Contents

1 Introduction

To start with, we explain synthesis which can be stated as the problem of finding a completion for a partial program so that it complies with a given specification. This kind of problems can be encoded with context-free games and we therefore proceed with those presented in [HMM16]. We sketch the structure and content of the remainder of the thesis to conclude the introduction.

1.1 Synthesis

Due to the application of positive Boolean formulas in synthesis, we explore different operations for these formulas and focus on the most important one: implication checks like $F \Rightarrow G$. We analyze their complexity and present different approaches to handle them in practice.

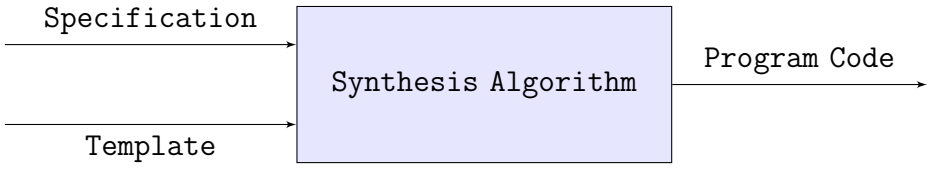


Figure 1: Synthesis: The goal is to find an instantiation for a given template so that the resulting program satisfies the given specification.

Terms and Problem We give some more intuition to the terms and concepts used in Figure 1 that illustrates the idea of synthesis. A template is a fragment of program code with gaps, i.e. there are missing expressions, so that the code is actually incomplete. An instantiation of a template is a possible completion in which all missing expressions are specified. So we state the problem of synthesis as follows:
Synthesis problem Given a program template T and specification φ. Is there an instantiation $T@i$ of T satisfying φ ?

Advantages By now, the question might raise why it is advantageous to describe the behaviour of a program by a specification instead of its code. The goal of synthesis is to minimize the effort of designing low-level components for software- and hardware-designers [Kre98]. Instead of dealing with low-level concerns, designers should be able to develop high-level specifications rather including the desired behaviour of the product than design details. Thereby, one can use the specification, instead of the code, for program verification leading to more reliability which is crucial for the frequent application in safety-critical systems.

From templates to CFG If a template is missing a condition for an if-then-else-statement, we have to determine an appropriate one so that the specification is satisfied for any execution. In order to model this behaviour, we explain the

difference between two kinds of non-determinism with the two code examples given in Figure 2. The uncontrollable non-determinism is also called demonic while

Demonic non-determinism

```
proc F()
    if(x == 0)
        G()
    else
        H()
```

F → read(x,0)G | read(x,1)H

Angelic non-determinism

```
proc F()
    if ???
        G()
    else
        H()
```

F → G | H

Figure 2: Demonic non-determinism is uncontrollable while angelic non-determinism is controllable.

the controllable non-determinism is called angelic. To model these two kinds in the grammar, every non-terminal is owned by one of two players, i.e. prover or refuter. As indicated by the names, prover's goal is to prove that every possible execution satisfies the specification whereas refuter tries to find an execution which violates the specification. Intuitively, the controllable non-determinism is handled by prover since he wants to find a strategy for the program to satisfy the specification. Refuter deals with the uncontrollable non-determinism as it is sufficient to have one bad execution to violate the specification. Therefore, non-terminals representing demonic non-determinism are owned by refuter whereas those representing angelic non-determinism are dedicated to prover. On the one hand, context-free grammars are a good approximation to describe the syntax of most programming languages and therefore we can use them to model the control flow of templates. On the other hand, we can describe valid executions by a regular specification and therefore queries like $\mathcal{L}(C) \subseteq \mathcal{L}(A)$, i.e. inclusion of a context-free language in a regular language, are essential to verification. One could use such queries in an iterative refinement loop to amend the regular specification in order to learn about the semantics of the program. This allows us to handle real programs.
The whole concept of controllable and uncontrollable non-determinism is not restricted to software but can also be used for hardware circuits. As before, the input is divided into a controllable and an uncontrollable part and the output has to meet a regular specification. As the controllable part can depend on the uncontrollable one, it does not suffice to check any combinatorial combination. So overall, the framework is similar.
Next, we give a brief introduction to the summary technique for context-free games from [HMM16].

1.2 Summary Technique for Context-free Games

Context-free Games When we move from queries like $\mathcal{L}(C) \subseteq \mathcal{L}(A)$ to context-free games, we convert this inclusion query to a strategy synthesis for an inclusion game. If the inclusion holds, prover has a winning strategy as he can force any play to be either infinite or end in a word in the regular language. The partition of non-terminals we mentioned before is given by an ownership partitioning of the non-terminals in the grammar whereas the regular language is represented by a non-deterministic finite automaton. For the derivation of words in the context-free grammar, we assume that only rules for the leftmost non-terminal in a sentential form are applied. This is called left-derivation and basically coincides to executing the program sequentially. So the decision which rule to apply is given by the strategy of the owner of the leftmost non-terminal. Overall, this induces a game arena of sentential forms in which the owner of the leftmost non-terminal takes turn. In general, inclusion in the regular language is the winning condition. To be precise, prover wants to prove that every derivable word is included in the regular language whereupon refuter wants to disprove the inclusion by finding a derivable word which is not included in the regular language.

From Plays to Formulas All plays starting in a non-terminal yield a (typically infinite) tree where inner nodes with their logical connectives represent the behaviour of the two players. The goal is to represent these plays finitely by positive Boolean formulas. We play the game from refuter's perspective and it suffices for refuter to have one good move for every possible rule application of prover. This observation is modelled by logical operators, as $\vee$ is true if one of its parts is true and therefore represents one good move, while $\wedge$ only evaluates to true if all parts are true and thus corresponds to every possible application of prover. To represent a play, the propositions of such formulas would be the set of terminal words. Because of the infiniteness of this domain, we only consider the state changes in the NFA that are induced by a terminal word to have a finite domain. Hence, these relations are the atomic propositions of our formulas.

Fixed Point Iteration To compute all plays starting in a non-terminal X, we use a fixed point iteration on the system of equations given by the production rules of the grammar. Actually, we compute the fixed-point on any non-terminal as these are needed to compute the plays starting in X. This system comprises all possible actions of both players and therefore the shape of the tree. Considering the iteration, we start with *false* for any non-terminal since *false* represents a strategy of prover to enforce an infinite derivation. During the computation, we get the resulting formula for a sentential form $\alpha\beta$ by composing the two formulas for α and β appropriately. With the left-derivation, every play in the resulting formula is given by a maximal play in α followed by a play in β. For the Kleene iteration, we use implication as partial order so that it will not be harder to satisfy a newly created formula compared to its predecessor for the same non-terminal. By Kleene's fixed-point theorem, we reach the least fixed point of the given systems of equation as we started with the

bottom *false* of the domain. Then, we check whether the final formula for the start symbol of the grammar is rejected by the automaton. Therefore, boxes are mapped to truth values accordingly and we evaluate the formula.

Implication Checks Furthermore, formulas are used up to logical equivalence. This is sufficient for termination because of the finiteness of this domain. So every time we compute a new formula, we know that the former one implies the new one by the Kleene iteration. Therefore, it suffices to check whether the new one implies the old one as well. These are exactly the implication checks we analyze in this thesis. We show i.a. that checking entailment is co-NP-complete even if satisfiability is trivial for positive Boolean formulas. Intuitively, it is harder than the satisfiability problem as implications introduce negations to our formulas.

1.3 Outline

The remainder of this thesis is structured as follows. In Chapter 2, we present the different concepts from the introduction formally. After the definition of non-deterministic finite automata, boxes and context-free grammars, we proceed with the positive Boolean formulas and present some of their properties like monotonicity. During the explanation of the summary technique for context-free games from [HMM16], we lift relational composition to our formulas and explore its behaviour.

Chapter 3 analyzes the complexity of entailment checking. We show the problem to be co-NP-complete, i.e. it is in co-NP and co-NP-hard. For the reduction, we present a way to transform any 3SAT instance with non-mixed clauses to checking non-entailment of two positive Boolean formulas. There are some special cases for formulas in CNF or DNF for which the implication checks are polynomial and hence we present these algorithms and prove their correctness.

In Chapter 4 and 5, we introduce two kinds of algorithms to solve the implication checks of positive Boolean formulas with and without compositions.

Chapter 4 introduces a sequent calculus with some common inference rules handling conjunctions and disjunctions as well as inference rules for composition which are used to handle the relational compositions in formulas. We prove the presented calculus to be sound and complete and show a bound on the number of applications of inference rules for conjunction and disjunction to prove a sequent to be a tautology.

In Chapter 5, we present an algorithm in which we exploit the monotonicity of positive Boolean formulas. First, we only consider formulas without compositions. Afterwards, we propose an extension to handle compositions symbolically and refine them if necessary.

Chapter 6 discusses possibilities to exploit the process of iterations. At first, we show that unchanged parts in the left formula can be omitted if both formulas are

disjunctions whereas those in the right formula can be omitted if both formulas are conjunctions. Then, we show that we cannot easily split compositions and handle them seperately, even if we exploit the monotonicity of the Kleene iteration.

We have implemented the summary technique for context-free games from [HMM16] and explain the structure of our tool in Chapter 7. Moreover, we describe how the theory about formulas without compositions is realised in practice and point to the challenges for versions with compositions.

Consequently, Chapter 8 gives insights to our benchmarking results. First, we explore the benefits of different preprocessing procedures that have been proposed in previous chapters. Second, we compare the approaches for formulas without compositions by embedding these techniques in a fixed-point iteration.

Finally, Chapter 9 concludes the thesis and reveals possibilities for future work.

2 Preliminaries

2.1 Notation and Definitions

At first, the common concepts like nondeterministic finite automata and context-free grammars which are used in the summary approach in [HMM16] are introduced. Then, we present the domain of positive Boolean formulas and prove some of its properties. Furthermore, we explain the summary technique with its fixed-point iteration and extend established definitions according to the given requirements.

Nondeterministic Finite Automaton

A *nondeterministic finite automaton (NFA)* is a tuple $A = (T, Q, q_0, Q_F, \rightarrow)$ with a finite alphabet T, a finite set of states Q, an initial state $q_0 \in Q$ and the transition relation $\rightarrow \subseteq Q \times T \times Q$ for which we adopt the notation of $q \xrightarrow{a} q'$ for $(q, a, q') \in \rightarrow$. Abbreviating the notation for several state changes induced by a word $w \in T^*$ leads to the following extension: $q \xrightarrow{w} q'$. This means that there is a sequence of states $q, \ldots, q'$ for which transitions labelled by the corresponding terminal between consecutive states exist so that the sequence of terminals build w. The language $\mathcal{L}(A)$ represented by an NFA is the set of all words w for which $q \xrightarrow{w} q_f$ for some $q_f \in Q_F$ exists. In general, any language $\mathcal{L}(A)$ for some NFA A is regular, i.e. it is definable by a regular expression as defined in [HU79]. The complement of $\mathcal{L}(A)$ is the set of all words for which such a transition sequence does not exist, denoted by $\overline{\mathcal{L}(A)} = T^* \setminus \mathcal{L}(A)$. From now on and without further mentioning, A denotes an NFA and T is a set of terminals. Furthermore, let $a, b \in T$ be terminals while $w, v \in T^*$ denote words over T.

As seen before, terminals can induce several state changes in an NFA. These changes for w can be represented by a relation $\rho_w = \{(q, q') \mid q \xrightarrow{w} q'\}$ called *box*. The box of a word $xy \in T^*$ can be constructed by composing the relations for the two shorter words: $\rho_{xy} = \rho_x \circ \rho_y$. As the relational composition is associative and $\rho_\varepsilon = \{(q, q) \mid q \in Q\}$ is the neutral element respective to composition, we call $TM(A) = (\{\rho_w \mid w \in T^*\}, \circ)$ the *transition monoid* of A. To distinguish boxes of words $w \in \mathcal{L}(A)$ and $w' \in \overline{\mathcal{L}(A)}$, we call a box ρ_w *rejecting* iff there is no path from the initial state to any final state, i.e. $(q_0, q_f) \notin \rho_w$ for any $q_f \in Q_F$. Considering this property, the NFA induces a mapping from boxes to $\{\mathit{false}, \mathit{true}\}$, i.e. $\rho_w \mapsto \mathit{true}$ iff ρ_w is rejecting.

Example 1. The automaton in Figure 3 is the first part of our running example and will be used for explanations consistently. It accepts all words inducing a run ending in state q_1, i.e. all words in the regular language $\mathcal{L}(a^*ab^*)$. Generally speaking, the boxes are the induced relations. For instance, the neutral element ρ_ε is the box with horizontal arrows only. We can observe that any relation does not change by concatenating ρ_ε and in general, we can find the result by linking up the boxes and collect all possible ways to get from one state to any other. Consider ρ_{ab} for instance. Initially, we can use the arrow from q_0 to q_1 and proceed with the only one in ρ_b. Following the arrow from q_0 to q_0 does not lead to an arrow in the result since there

are no connections between q_0 and any other state in ρ_b. Thus, there is only one arrow in ρ_{ab} since there is only one possible state change induced by the terminal word ab.

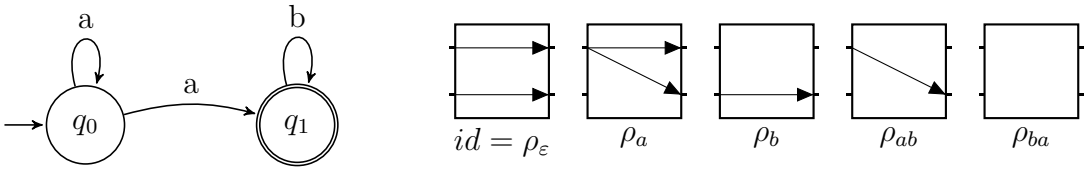


Figure 3: NFA A with two states that accepts $\mathcal{L}(a^*ab^*)$ with its corresponding boxes.

Context-free Grammar

Just like regular languages can be modeled by NFAs, there is a common way to represent context-free languages. A *context-free grammar (CFG)* is a tuple $C = (N, T, P)$, where N is a finite set of non-terminals and T is a finite set of terminals such that $N \cap T = \emptyset$. In order to define the production rules P, we call all combinations of non-terminals and terminals the set of sentential forms $\vartheta = (N \cup T)^*$. $P \subseteq N \times \vartheta$ is a finite set of production rules mapping every non-terminal to at least one sentential form and we adopt the notation of writing $X \to \eta$ instead of $(X, \eta) \in P$ here as well. In general, grammars can also be used to model languages different from the context-free ones. To represent regular languages, we restrict the productions to be $P \subseteq N \times \{aM \mid a \in T, M \in N\}$ whereas for context-sensitive languages, we could have terminals on the left side of a production as well.
Considering an *ownership partitioning* $N = N_\bigcirc \uplus N_\square$ of the set of non-terminals, each $X \in N_\bigcirc$ is dedicated to player $\bigcirc$ and conversely $Y \in N_\square$ to player $\square$. In the following example, we demonstrate how words can be produced by a context-free grammar.

Example 2. Consider the following context-free grammar:

$$C = (N, T, P) \text{ with } N = \{X_\bigcirc, Y_\square\}, T = \{a, b\}, \text{ and}$$
$$P = \{X_\bigcirc \to X_\bigcirc Y_\square \mid \varepsilon,\ Y_\square \to X_\bigcirc b \mid a\}.$$

We show a left-derivation from $X_\bigcirc$ where we omit the symbols indicating the ownership and ε's after one step:

$$X \to XY \to XYY \to \varepsilon YY \to XbY \to \varepsilon bY \to bXb \to bXYb \to b\varepsilon Yb \to bab.$$

Positive Boolean Formulas

One of the crucial points of the summary technique from [HMM16] is the usage of Boolean formulas for the sentential forms generated by the CFG with ownership partitioning. We only explain the intuition behind the construction, but we will explore it in more detail when we explain the summary technique in the next section.

One can construct a (typically infinite) tree where conjunctions in inner nodes are used to express the options of prover whereas disjunctions represent the options of refuter. As there often occur multiple right sides for the same non-terminal in a CFG and the same part can reoccur after a step of the iteration, it is advantageous to adjust the typical binary representation of operators to a representation with sets. After the explanation of the summary technique, we will also give some more reasoning to emphasize the advantages of this representation.

Definition 1. The set PBF_A of *Positive Boolean Formulas over the transition monoid of A* consists of *true*, *false* and all formulas defined by the following EBNF:

$$F ::= \rho_w \mid \bigwedge_{1 \leq i \leq n} F_i \mid \bigvee_{1 \leq i \leq n} F_i$$

where $w \in T^*$ is some word and $1 < n \in \mathbb{N}$ in every case so that operations actually do matter as they could be omitted otherwise. The unsatisfiable formula is represented by *false* whereas *true* represents the formula which is satisfied by any assignment as there is no way to construct these by the EBNF.

To include *false* and *true* in conjunctions and disjunctions, we handle them on a syntactic level as follows:

$$\bigwedge_{1 \leq i \leq n} F_i = \mathit{false} \text{ if } \exists i : F_i = \mathit{false}$$
$$\bigvee_{1 \leq i \leq n} F_i = \mathit{true} \text{ if } \exists i : F_i = \mathit{true}.$$
$$\bigwedge_{1 \leq i \leq n} F_i = \bigwedge_{1 \leq i \leq n \wedge F_i \neq \mathit{true}} F_i$$
$$\bigvee_{1 \leq i \leq n} F_i = \bigvee_{1 \leq i \leq n \wedge F_i \neq \mathit{false}} F_i$$

So verbally, if there is *false* in a conjunction, the whole conjunction is *false* and analogously if there is *true* in a disjunction, the whole disjunction is *true*. In a conjunction, *true* can be ommited whereas *false* can be omitted in disjunctions.

Evaluation We are able to evaluate formulas $F \in PBF_A$ for the assignment induced by A, which was defined by the property of rejecting, but we are actually not missing much to evaluate arbitrary assignments as well. In order to know the variables of F which have to be assigned, the set of ground symbols $GS(F)$ is defined as any ρ_w occuring in F. As mentioned only superficially before, an assignment $\varphi \subseteq GS(F)$ for F is the set of ground symbols assigned to *true*. It is also common to denote an assignment as a mapping $\{\rho_w \mid w \in T^*\} \to \{\mathit{true}, \mathit{false}\}$. The former version simplifies the definition of monotonicity whereas the latter is preferable for different definitions and lemmata. Hence, we use the two versions interchangeably. We also use the more convenient notation $\varphi(\rho_v)$ for $\varphi(\rho_v) = \mathit{true}$ and $\neg\varphi(\rho_v)$ for $\varphi(\rho_v) = \mathit{false}$. Furthermore, the conversion from Boolean values *true* and *false* to

1 respectively 0 is widely adopted and hence we will use this implicit conversion to simplify statements.
Given an assignment φ, the *evaluation* of $F \in PBF_A$ is defined as follows:

- $\varphi(\bigwedge_{1\leq i\leq n} F_i) = min_{1\leq i\leq n}(\varphi(F_i))$
- $\varphi(\bigvee_{1\leq i\leq n} F_i) = max_{1\leq i\leq n}(\varphi(F_i))$

Considering the time complexity of the evaluation process, one can observe that the time and space requirements are linear in the size of the formula. For this lemma, the well-known $O-$notation for complexity is used as defined in [Sip05].

Lemma 1. *Given an assignment φ and a formula $F \in PBF_A$ of size n, computing the evaluation $\varphi(F)$ is in $O(n)$ regarding time and space complexity.*

Proof. The formula can be represented by a finitely branching syntax tree labelled by ground symbols and logical connectives. Such a tree can be traversed by postorder, i.e. visit all children of a vertex and then itself. During the traversal, there are some additional steps needed to get the result:
A subtree of the syntax tree is a subformula and we represent its evaluation result in its root. Therefore, the vertices are additionally labelled by a unique index to store those intermediate results of any subformula in a Boolean array A. If a vertex is visited by the traversal, apply the following method. Given a vertex v indexed by i, there are several cases:

- v is a ground symbol: set $A[i]$ to $\varphi(G)$,
- v is a logical connective, i.e. either $\wedge$ or $\vee$. As the tree is traversed in postorder, all children of v have been visited before and their evaluation results are stored in A. Let C be the indices of all children of v.
  If v is labelled by $\wedge$, $A[i] = min\{A[i] \mid i \in C\}$.
  If v is labelled by $\vee$, $A[i] = max\{A[i] \mid i \in C\}$.

Returning $A[index(F)]$ gives the desired result.
The space and time complexity of postorder is $O(n)$. The Boolean array A has one entry for every vertex in the tree so the space complexity remains linear. As we compute the minimum respectively maximum for every vertex at most once, the time complexity of the extra steps is also linear. Altogether, this yields linear time and space complexity. □

Consequences of the Absence of Negation Succeeding the definition of syntax and semantics of PBF_A, we explore the effects of the absence of negation. These observations do not rely on the fact that the ground symbols are boxes of A but apply to any domain of positive Boolean formulas. Firstly, checking satisfiability of formula F, i.e. whether there is an assignment for which F is evaluated to *true*, is trivial since it is admissible to check the assignment $\varphi = GS(F)$. Secondly, if a formula $F \in PBF_A$ is satisfied by an assignment φ, i.e. it is evaluated to *true*, F will be satisfied by any other assignment $\varphi' \supseteq \varphi$. Analogously, the inverse also

holds. If some formula is not satisfied by an assignment, it will not be satisfied by any smaller assignment which is a subset of the former one. These observations are stated formally in the second lemma.

Lemma 2 (Monotonicity of Positive Boolean Formulas)**.**
For any $F \in PBF_A$ and $\varphi \subseteq \varphi' \subseteq GS(F)$, $\varphi(F) \leq \varphi'(F)$.

Proof. The proof is by an easy case analysis. Let $\varphi \subseteq GS(F)$ be any assignment.
If $\neg\varphi(F)$, the property holds for any $\varphi' \supseteq \varphi$ trivially.
If $\varphi(F)$, it is straightforward to check by structural induction on the different logical connectives that $\varphi(G) \leq \varphi'(G)$ holds for any subformula of F. □

Implication Since implication checks play a major rule in the overall approach, we recap the common definition of implication. For $F, G \in PBF_A$, F implies G, denoted by $F \Rightarrow G$, iff $\forall\varphi \subseteq GS(F) \cup GS(G)$: if $\varphi(F)$, then $\varphi(G)$. We certainly could include implication as a syntactic construct, but this secretly introduce negations to our formulas as they can be transformed in the following way: $F \Rightarrow G \models\!\!\!\models \neg F \vee G$ [1]. However, since we only need them when we check entailment, we refrain from doing this. Hence, these single implications will be displayed explicitly. To clarify this, the terms implication and entailment are equivalent in this context and therefore used interchangeably.

2.2 Summary Technique for Context-Free Games

Thinking of the synthesis of recursive programs again, we would like to run the execution sequentially, i.e. at first the effect of a called method is computed and afterwards we continue with the remainder of the program. Therefore, we use *left-derivation* $\Rightarrow_L$ (applying a rule for the leftmost non-terminal) to model this behaviour. It is worth mentioning that the usage of left-deriviation is not restrictive in any way, i.e. for any derivation, there is a left-derivation leading to the same terminal word [HU79].
For instance, consider $X_{\bigcirc}Y_{\square}$, we do not look at $Y_{\square}$ before any non-terminal on its left side has disappeared by the application of production rules. Thus, for a sentential form ϑ, the ownership depends on the leftmost non-terminal in ϑ, i.e. $X_{\bigcirc}Y_{\square}$ is owned by $\bigcirc$. So the set of all sentential forms can be split into the ones owned by player $\bigcirc$ and the ones dedicated to player $\square$: $\vartheta = \vartheta_{\bigcirc} \cup \vartheta_{\square}$.

Definition 2. Let $G = (N_{\bigcirc} \cup N_{\square}, T, P)$ be a CFG with ownership partitioning. The *arena induced by* G is the directed graph $(\vartheta_{\bigcirc} \cup \vartheta_{\square}, \Rightarrow_L)$.

A *play* $p = p_0 p_1 \ldots$ is a finite or infinite path in the arena where $p_i \Rightarrow_L p_{i+1}$ for all positions. If it is finite, the path ends in a vertex denoted $p_{last} \in \vartheta$. A path corresponds to a sequence of left-derivations, where for each leftmost non-terminal the owning player selects the rule that should be applied. A play is called *maximal*

[1] $\models\!\!\!\models$ is the bidirectional consequence operator, i.e. both sides evaluate to the same value for every assignment.

if its path has infinite length or if the last position is a terminal word, i.e. there are no derivations to apply anymore. The goal is to decide whether refuter can force a (maximal) play to end in a terminal word rejected by A. We define this intuition as follows:

Definition 3. The *inclusion game* and the *non-inclusion game* with respect to A on the arena induced by G are defined by the following winning conditions. A maximal play p satisfies the *inclusion winning condition* if it is either infinite or we have $p_{last} \in \mathcal{L}(A)$. A maximal play satisfies the *non-inclusion winning condition* if it is finite and $p_{last} \in \overline{\mathcal{L}(A)}$.

These two games are complementary, i.e. for every maximal play exactly one of the winning conditions holds. From now on, the symbol $\bigcirc$ indicates player refuter trying to produce a terminal word rejected by A, which is a reachability condition. In contrast, player $\Box$ tries to prove inclusion and does not have to enforce termination of the play, which represents a safety condition. Prover only has to remain in his winning region whereas refuter has to enforce termination and hence his strategy is harder. We therefore explain the succeeding construction from refuter's view.

Relational Compositions for Formulas

We would like to represent all plays from a non-terminal by a formula. So far, PBF_A is not powerful enough as we are not able to handle the concatenation of formulas occuring during the application of production rules of the grammar. We therefore extend the set of formulas by introducing relational composition for formulas which we only defined for boxes so far. At first, we define the syntactic set of composed formulas and then explore the behaviour of the new operator. Similarly to the set representation of disjunctions and conjunctions, we use lists for the compositions of formulas.

Definition 4. The set $CPBF_A$ of *Composed Positive Boolean Formulas over the transition monoid of A* consists of *true*, *false* and all formulas defined by the following EBNF:

$$F ::= \rho_w \mid \bigwedge_{1 \leq i \leq n} F_i \mid \bigvee_{1 \leq i \leq n} F_i \mid F_1; \ldots; F_n$$

where $w \in T^*$ is some word and $1 < n \in \mathbb{N}$ in every case so that operations actually do matter as they could be omitted otherwise. Analogously to the definition of PBF_A, *false* represents the unsatisfiable formula whereas *true* the formula which is satisfied by any assignment. To handle them, the same rules as in PBF_A apply.

Remark 1. We included *true* in our sets of formulas for the sake of completeness. Nevertheless, *true* will not appear in our computation which we explain in the next section. Though it is possible to adapt this technique in order to use *true*, this does not change the result or the method profoundly. However, in this case *false* never appears and generally speaking, *false* and *true* never occur simultaneously.

Therefore, we only show how to handle *false* on a syntactic level for compositions and take over the handling we introduced for conjunctions and disjunctions:

$$F_1; \ldots; F_n = false \text{ if } \exists i : F_i = false.$$

Consequently, we explore how to lift the behaviour of relational compositions to formulas. Consider a maximal play $\alpha\beta$ which can be split into two phases. Because of the left-derivation the sentential form representing the state of the game is $w\beta$ at some point. As there are no more derivations in w, the remaining play $w\beta$ is determined by the sentential forms of the second play β prefixed by w.
Initially, let ρ_w denote the box for w and G some formula representing all plays starting in β. As described before, all boxes in G are prefixed by w to get a formula for the whole sentential form. Thus for each ρ_v in G, $(\rho_w; \rho_v)$ is computed using A and the results are logically connected by the operator of G again.
Finally, let F be a formula representing all plays starting in α. All plays in F are represented by boxes ρ_x. As any play starting in β, represented by G, succeeds any play represented in F, we append the plays from β to any box ρ_x in F: $\rho_x; G$. Similarly, these results are connected by the logical operator of F. Then, $(\rho_x; G)$ is handled as described before.
When we deal with compositions, we combine syntactically different formulas that only differ the number of compositions that have been removed. As the process of dealing with compositions combines syntactically different formulas that only differ in the number of removed compositions, we define a congruence relation $\cong$ on $CPBF_A$. Prior to its definition, we present a short example for the process itself.

Example 3. We show how to remove all compositions in $[(\rho_a \wedge \rho_b) \vee \rho_\varepsilon]; (\rho_a \wedge \rho_b)$.

$$\begin{aligned}
& [(\rho_a \wedge \rho_b) \vee \rho_\varepsilon]; (\rho_a \wedge \rho_b) \\
\overset{\text{(split 1st formula)}}{\cong} \; & [(\rho_a \wedge \rho_b); (\rho_a \wedge \rho_b)] \vee [\rho_\varepsilon; (\rho_a \wedge \rho_b)] \\
\overset{\text{(split in front term \& exploit } \rho_\varepsilon; \rho = \rho)}{\cong} \; & [(\rho_a; (\rho_a \wedge \rho_b)) \wedge (\rho_b; (\rho_a \wedge \rho_b))] \vee [\rho_a \wedge \rho_b] \\
\overset{\text{(split both 2nd formulas in front term)}}{\cong} \; & [\rho_{aa} \wedge \rho_{ab} \wedge \rho_{ba} \wedge \rho_{bb}] \vee [\rho_a \wedge \rho_b] \\
\overset{(\rho_{bb} = \rho_b)}{=} \; & [\rho_{aa} \wedge \rho_{ab} \wedge \rho_{ba} \wedge \rho_b] \vee [\rho_a \wedge \rho_b]
\end{aligned}$$

Definition 5. The *Relational Composition* over $CPBF_A$ is defined by the congruence relation $\cong$ which is the smallest congruence satisfying the following rules.
For $n, m \in \mathbb{N}$, $v, w \in T^*$ and $\Diamond \in \{\vee, \wedge\}$:

- $\rho_v; \rho_w; F_1; \ldots; F_n \cong \rho_{vw}; F_1; \ldots; F_n$
- $\rho_v; (\Diamond_{1 \le i \le m} G_i); F_1; \ldots; F_n \cong \Diamond_{1 \le i \le m} (\rho_v; G_i; F_1; \ldots; F_n)$
- $(\Diamond_{1 \le i \le m} G_i); F_1; \ldots; F_n \cong \Diamond_{1 \le i \le m} (G_i; F_1; \ldots; F_n)$

For all rules, the remainder $F_1, \ldots, F_n$ is untouched and it is empty for $n = 0$.

With this congruence relation, we can use two congruent formulas G and G' interchangeably. E.g. we can subsitute one by another in a bigger context: For some formula F, if $G \cong G'$, then $F \wedge G \cong F \wedge G'$.

Because of the list representation for compositions, we could actually combine the second and the third rule, but we refrained from doing this in order to illustrate the analogy between these rules and the behaviour we explained for plays before. The first rule states that two boxes can be composed whereas the second shows that boxes can be shifted into the next formula. This process coincides with the plays where each play in the second formula is prefixed by the terminal word. In the last rule, the first formula is split and this coincides with the step where every play of the first formula is appended by the remainder of the composition.

By those rules, the domain $CPBF_A$ can be split into disjoint congruence classes which can be represented by a canonical representative. For the sake of uniqueness, we define this to be the only formula without compositions in this class. It exists as we can handle any occurrence of compositions by one of the rules above. It is unique as at most one of the rules can be applied to a formula. Hence, we can find a congruent formula in PBF_A for any formula in $CPBF_A$.

Fixed Point Iteration

We are able to represent all plays from a non-terminal with the extended set of formulas $CPBF_A$ where boxes are propositions. As the game is played from one player's perspective, his goal is to have at least one good move, i.e. a rule application so that he remains in his winning region, for every possible sentential form owned by him. Those are also built by the opponent's choices so that the mentioned goal includes the requirement of having a good answer to any move by the opponent. So for a procedure summary, one can construct a (typically infinite) tree where conjunctions in inner nodes are used to express the options of prover whereas disjunctions represent the options of refuter. The idea is to iterate on those formulas to explore which player has a winning strategy.

Finiteness To guarantee termination though, finiteness of these trees is crucial. It is sufficient to have a finite number of propositions since there are only finitely many logically equivalent positive Boolean formulas over a finite set of propositions. This is straightforward to see as there are exactly 2^n different Boolean functions for n propositions. In our domain, maximal plays are represented by terminal words. While the number of the latter is infinite, we can factorize them into their equivalence classes over A by using the boxes defined before. To be able to refer to those equivalence classes, we define the language of a box ρ: $\mathcal{L}(\rho) = \{w \mid \rho_w = \rho\}$. Obviously, there can be boxes for which this language is empty.
The maximal number of boxes for A with n states corresponds to the maximal number of binary relations for a set M of size n. Let $C = (M \times M)$ with size $n \cdot n$. Each subset of C corresponds to exactly one relation, so counting them leads to the

desired result: $2^{n \cdot n}$ which is finite.
So we can use $CPBF_A/_{\Leftrightarrow}$ as our finite domain and check equality by checking logical implication.

Kleene Iteration For Kleene iteration, one usually needs a partial order. Intuitively, $F \leq G$ for two formulas $F, G \in CPBF_A/_{\Leftrightarrow}$ if it is easier for refuter to win the play represented by G. Logically, this means $F \leq G$ iff $F \Rightarrow G$. So we should check whether $\Rightarrow$ is a partial order, i.e. if it is reflexive, transitive and antisymmetric. Reflexivity and transitivity are trivial. In contrast, the implication is not antisymmetric for the general domain PBF_A. Consider two logically equivalent formulas $K, L \in PBF_A$, $K \Rightarrow L$ and $L \Rightarrow K$ hold but they are not the same. This is another reason why we use $CPBF_A/_{\Leftrightarrow}$. There, $K \in [L]_{\Leftrightarrow}$ and thus antisymmetry holds. So $\Rightarrow$ yields a partial order on $CPBF_A/_{\Leftrightarrow}$. Furthermore, the function to compute the fixed-point has to be monotone. We therefore investigate our three different operators. Monotonicity was proven for composition-free formulas in Lemma 2, so it is sufficient to recall a lemma from [HMM16] without proof where monotonicity is proven for the relational composition.

Lemma 3. *If $F \Rightarrow F'$ and $G \Rightarrow G'$, then $F;G \Rightarrow F';G'$.*

We compute a fixed point over a number of formulas, i.e. for every non-terminal in the CFG. So for each non-terminal X, one step should compute a new formula by using its right-hand side in the grammar. The outermost logical connective of the formula for a non-terminal X depends on its owner, i.e. $\wedge$ for prover and $\vee$ for refuter. Technically, it is a monotonic function $f_X : (CPBF_A/_{\Leftrightarrow})^N \to CPBF_A/_{\Leftrightarrow}$ which takes a vector of formulas (one for each non-terminal) and computes a new formula for X. Lifting this to all non-terminals yields a single function:

$$f : ((CPBF_A/_{\Leftrightarrow})^{|N|} \to (CPBF_A/_{\Leftrightarrow})^{|N|}).$$

We explain the construction with an example.

Example 4.
We show the iteration process for the automaton and the grammar introduced in Example 1 and 2.

Nr.	$X_{\bigcirc}$	$Y_{\Box}$
0	*false*	*false*
1	ρ_ε	*false*
2	ρ_ε	$\rho_a \wedge \rho_b$
3	$(\rho_a \wedge \rho_b) \vee \rho_\varepsilon$	$\rho_a \wedge \rho_b$
4	$(\rho_a \wedge \rho_b) \vee \rho_\varepsilon$	$\rho_a \wedge ([(\rho_a \wedge \rho_b) \vee \rho_\varepsilon]; \rho_b)$ $\Rightarrow \rho_a \wedge \rho_b$ (by Ex.6)

Implication Checks As explained before, termination is only enforced by using $CPBF_A/_{\Leftrightarrow}$ instead of $CPBF_A$. Thus checking syntactic equivalence is not sufficient to find a fixed-point but we need to check whether two formulas are semantically

equivalent. As we use Kleene iteration, we know that $F \Rightarrow f(F)$ so we only need to check whether $f(F) \Rightarrow F$ to show $F \Leftrightarrow f(F)$. Computing new formulas with logical operators and the relational composition is not very costly but checking implication can be. As this operation is used very frequently and is central to the presented approach, we investigate the complexity of entailment checking and different approaches to check implication for composition-free and symbolic formulas, i.e. formulas containing compositions, in the remainder of this thesis.

Remark 2. After we explained the technical part in more detail, we come back to the reasoning why we use set representation for conjunctions and disjunctions and lists for compositions.
To explain advantages of the set representation, consider the context-free grammar again. Consider a non-terminal N which occurs on the right side of its own production rule. We need to recompute the formula if at least one part of its right side changed. Assume that the formula for N did not change. If we used a binary representation, the formula would reoccur on some deeper level of the syntax tree, connected by the same logical operator, and thus unnecessary for the semantic meaning of the formula. Furthermore, two parts on the right side of a production can result in the same formula. It might seem we could neglect these overheads but these can increase enormously as we are dealing with a fixed-point iteration.
For the use of lists for compositions there are more practical reasons. For the application of the rules for relational composition, one of several conditions has to hold: First, the first two components are two boxes which can simply be composed. Second, the first component is a box and the second one is a formula with a logical operator. Third, we are able split the first component of a composition if it is a formula with a logical operator.
So we have to ensure that the operator in the next level is not a compositional relation. Therefore, we decided to use this list representation. We are aware of the fact that we could look for the actual first and second component of the composition every time but for the sake of simplicity we take it for granted that the outermost operator of any component of a composition is different from the relational composition.
We could also assume that all formulas have alternating operators, i.e. that every component of a conjunction, disjunction or composition is not of the same kind as those could have been lifted otherwise. In contrast to the assumption about composition, we do not assume alternating operators. In practice, this can be advantageous but it is not significant for the succeeding theory.

3 Co-NP-Completeness

After underlining the importance of implication checks for the overall approach, we analyze the complexity of entailment checking for two formulas $F, G \in PBF_A$. The main result of this chapter is the co-NP-completeness of this problem which shows its hardness, unless P = NP. This has already been proven in [DG93] in which the authors introduce new variables for negated variables in order to reduce entailment checking of Boolean formulas to entailment checking of positive Boolean formulas. We take a different approach. First, we show membership in co-NP. Second, a reduction from 3SAT for non-mixed clauses to checking implication is constructed. The question might rise why the monotonicity shown before does not simplify the problem. As alluded to before, the implication secretly introduces negations on the left side and this impairs the monotonicity applying to positive Boolean formulas. In contrast, the monotonicity can be exploited for some combinations of formulas in CNF and DNF, so we present polynomial time algorithms for them and prove their correctness.

3.1 Complexity and Reductions

Considering time and space requirements, there are several classes of decidable problems. For this section, we use the well-known definition of NP, the class of problems decidable by non-deterministic Turing-machines in polynomial time, and its complement co-NP as well as the definition of polynomial reductions from [Sip05]. Let us shortly recap that a problem $A \subseteq \Sigma^*$ is polynomial reducible to $B \subseteq \Gamma^*$ if there is a total function, computable in polynomial time, with which instances of Σ^* can be transformed into instances of Γ^* so that $x \in A \Leftrightarrow f(x) \in B$.

3.2 Membership in co-NP

A problem is in co-NP if there is a non-deterministic algorithm of the following shape: One can guess a candidate for a counterexample, verify it in polynomial time and return false if it actually was a counterexample. An input is accepted if all branches, i.e. all guesses, return true and it is rejected if there is at least one branch yielding false.

Lemma 4 (Membership in co-NP)**.**
Checking Entailment for $F, G \in PBF_A$, i.e. whether $F \Rightarrow G$ holds, is in co-NP*.*

Proof. The result is proven by presenting a nondeterministic algorithm with time complexity $O(n)$.

- First, guess a candidate for a counterexample,
  i.e. an assignment $\varphi \subseteq (GS(F) \cup GS(G))$.
- Second, evaluate $\varphi(F)$ and $\varphi(G)$ in $O(n)$ by Lemma 1 and check whether F is satisfied and G is not satisfied by φ. If this is the case, we return false and if not, we return true.

If the implication does not hold, there is at least one counterexample. If we guess it, the algorithm returns false so the input is not accepted. In case the implication holds, there is no counterexample and therefore the algorithm returns true for all guesses. □

3.3 Co-NP-Hardness

Complement However, in our use case the technique of reduction is not used directly. We consider the complement of the problem instead, i.e. checking whether the implication $F \Rightarrow G$ does not hold, and denote it by its negation since we know that H is a tautology iff its negation $\neg H$ is unsatisfiable from the logic's point of view. We show co-NP-hardness by proving the complement to be NP-hard.

3SAT We consider an NP-hard problem and present a polynomial reduction which transforms one arbitrary instance into an instance of non-entailment checking. We use the NP-hard problem 3SAT with non-mixed clauses, i.e. in any clause all literals can either occur positively or negatively, which was shown to be NP-hard in [Sch78]. To be precise, they presented a list of properties so that an instance of 3SAT is in P iff it satisfies at least one of these properties and is NP-complete otherwise.

More General As the reduction works for any domain of positive Boolean formulas PBF, we use letters instead of boxes as variables to improve readability. We can substitute each letter by a box of A in order to transfer this kind of instances to our domain. Prior to going into details of the technical construction, we illustrate the idea with an example.

Example 5. Let $a, \ldots, e$ be any variables.

$$
\begin{aligned}
& (a \vee b \vee c) \wedge (a \vee d \vee e) \wedge (\neg a \vee \neg c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg e) \\
\overset{\text{(De Morgan)}}{\models\!\dashv} \quad & (a \vee b \vee c) \wedge (a \vee d \vee e) \wedge \neg(a \wedge c \wedge d) \wedge \neg(b \wedge c \wedge e) \\
\overset{\text{(De Morgan)}}{\models\!\dashv} \quad & (a \vee b \vee c) \wedge (a \vee d \vee e) \wedge \neg((a \wedge c \wedge d) \vee (b \wedge c \wedge e)) \\
\overset{(\neg\neg F = F)}{\models\!\dashv} \quad & \neg\neg[(a \vee b \vee c) \wedge (a \vee d \vee e) \wedge \neg((a \wedge c \wedge d) \vee (b \wedge c \wedge e))] \\
\overset{\text{(De Morgan)}}{\models\!\dashv} \quad & \neg[\neg((a \vee b \vee c) \wedge (a \vee d \vee e)) \vee ((a \wedge c \wedge d) \vee (b \wedge c \wedge e))] \\
\overset{(F \Rightarrow G \,\dashv\vdash\, \neg F \vee G)}{\models\!\dashv} \quad & \neg[((a \vee b \vee c) \wedge (a \vee d \vee e)) \Rightarrow ((a \wedge c \wedge d) \vee (b \wedge c \wedge e))]
\end{aligned}
$$

Lemma 5 (co-NP-hardness)**.** *Checking Entailment for $F, G \in PBF$, i.e. whether $F \Rightarrow G$ holds, is* co-NP*-hard.*

Proof. Guided by the construction illustrated in Example 5, the reduction will be stated formally.

Let $A = (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \wedge (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} \neg b_{ij})$. Hence, the same transformations are executed.

$$
\begin{aligned}
& (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \wedge (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} \neg b_{ij}) \\
\overset{\text{(De Morgan)}}{\models\!\dashv} \quad & (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \wedge (\bigwedge_{1\leq i\leq n} \neg \bigwedge_{1\leq j\leq 3} b_{ij}) \\
\overset{\text{(De Morgan)}}{\models\!\dashv} \quad & (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \wedge (\neg \bigvee_{1\leq i\leq n}\bigwedge_{1\leq j\leq 3} b_{ij}) \\
\overset{(F=\neg\neg F)}{\models\!\dashv} \quad & \neg\neg[(\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \wedge (\neg \bigvee_{1\leq i\leq n}\bigwedge_{1\leq j\leq 3} b_{ij})] \\
\overset{\text{(De Morgan)}}{\models\!\dashv} \quad & \neg[\neg(\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \vee (\bigvee_{1\leq i\leq n}\bigwedge_{1\leq j\leq 3} b_{ij})] \\
\overset{(F\Rightarrow G \models\!\dashv \neg F\vee G)}{\models\!\dashv} \quad & \neg[(\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}) \Rightarrow (\bigvee_{1\leq i\leq n}\bigwedge_{1\leq j\leq 3} b_{ij})]
\end{aligned}
$$

So for $F = (\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq 3} a_{ij}), G = (\bigvee_{1\leq i\leq n}\bigwedge_{1\leq j\leq 3} b_{ij}) \in PBF$, A can be reduced to $\neg(F \Rightarrow G)$ so that its complement is shown to be co-NP-hard. □

Altogether, this yields our first theorem.

Theorem 1.
Checking Entailment for $F, G \in PBF$, i.e. whether $F \Rightarrow G$ holds, is co-NP*-complete.*

CNF and DNF There is some regularity in the final formulas of our construction and therefore we shortly recap the frequently used conjunctive and disjunctive normal form. Even if we have omitted negations in our domain, we can observe that implications introduce them. Just like the implication itself, we will always state them explicitly, but we permit negations for the following definitions because we only use them for the remainder of this section. A *literal* is either ρ_w or $\neg\rho_w$ for some $w \in T^*$. A *clause* is a disjunction of literals whereas a *co-clause* is a conjunction of literals. A formula in *conjunctive normal form (CNF)* is a conjunction of clauses, frequently represented by a set of clauses. Analogously, a formula in *disjunctive normal form (DNF)* is a disjunction of co-clauses.

3.4 Special Cases

We have shown that checking $F \Rightarrow G$ for F in CNF and G in DNF is hard, unless $\mathsf{NP} = \mathsf{P}$. Conversely, polynomial algorithms exist for all remaining combinations of CNF and DNF. For the remainder of this section, let $F, G \in PBF$. An algorithm for F and G in CNF has already been presented in [HMM16] and [DG96]. The following lemma simply restates this result without proof.

Lemma 6. *For F in DNF and G in CNF, $F \Rightarrow G$ if and only if for every clause L of G there is a clause K of F such that the variables of K are a subset of the variables of L.*

Algorithms for the remaining two combinations have been stated in [DG96] without proofs, so they are restated and proven.

Lemma 7. *For F in DNF and G in CNF, $F \Rightarrow G$ if and only if every co-clause in F and every clause in G have at least one variable in common.*
Proof.

($\Leftarrow$) Consider an assignment φ for which F is evaluated to true. Then, there is at least one co-clause D in F which evaluates to true: $\varphi(D) = 1$. Therefore, all variables in D have to be assigned to true: $\forall d \in D : \varphi(d) = 1$. As each clause of G contains at least one of those d, all clauses evaluate to true. Thus, $\varphi(G) = 1$.

($\Rightarrow$) By contraposition, assume that there is a co-clause D in F and a clause C in G that do not have any variable in common. Consider the assignment $\varphi := \{d \mid d \in D\}$. As F is a disjunction of co-clauses and φ satisfies D, $\varphi(F) = 1$. As D and C do not share any variable, C is not satisfied by φ. Since G is a conjunction of clauses, $\varphi(G) = 0$. So altogether, the implication does not hold.

□

Lemma 8. *For F in DNF and G in DNF, $F \Rightarrow G$ if and only if for every co-clause D_F in F there is a co-clause D_G in G such that the variables of D_F are a superset of the variables of D_G.*
Proof.

($\Leftarrow$) Consider an assignment φ for which F is evaluated to true. Then, there is at least one co-clause D_F in F which evaluates to true: $\varphi(D_F) = 1$. There is a co-clause D_G in G whose variables are a subset of the variables of D_F. Therefore, $\varphi(D_G) = 1$ and $\varphi(G) = 1$.

($\Rightarrow$) By contraposition, assume there is a D_F in F and there is no D_G in G such that the variables of D_F are a superset of those in D_G. Let F' be the formula F without the co-clause D_F. Wlog, consider an assignment φ with $\varphi(F) = 1$ such that $\varphi(D_F) = 1$ and for $\varphi(F') = 0$. As there is no suitable subset for D_F, there is no co-clause in G which satisfies so that $\varphi(G) = 0$.

□

These results give rise to the question whether it is useful to handle compositions in implication checks or consider non-normalized formulas at all. As illustrated in Figure 4, removing compositions can increase the size of the instance multiplicatively while normalizing formulas can even be exponential [MRW05]. Thus, approaches for composed and non-normalized formulas are needed and therefore the main concern of the following chapters.

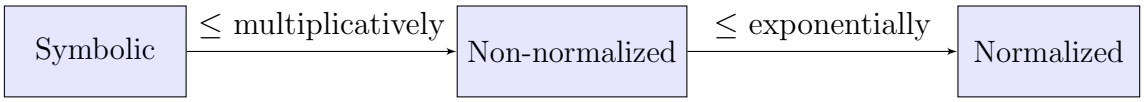


Figure 4: Increase in size: From symbolic formulas to non-normalized, the size can increase multiplicatively. Normalizing these formulas can even inflate the size exponentially.

4 Sequent Calculus

We propose an adapted version of the sequent calculus presented in [Bus98] for PBF_A which is able to handle compositions and prove it to be sound and complete. The sequent calculus was firstly introduced by Gentzen [1935] and is a flexible and elegant way for writing proofs.

4.1 Notation and Semantics

As indicated by the name, sequents are crucial for proofs in the sequent calculus. In detail, every line in a proof is called a *sequent* and has the following shape

$$A_1, \ldots, A_k \to B_1, \ldots, B_l$$

where $l, k \in \mathbb{N}$, $A_1, \ldots, A_k, B_1, \ldots, B_l \in CPBF_A$ and $\to$ is a special symbol, called the *sequent arrow*. The similarity between the sequent arrow and the symbol for implication is volitional as the sequent is supposed to mean

$$\bigwedge_{1 \leq i \leq k} A_i \Rightarrow \bigvee_{1 \leq j \leq l} B_j.$$

Consequently, we define a sequent $A_1, \ldots, A_k \to B_1, \ldots, B_l$ to be *valid* iff its corresponding formula $\bigwedge_{1 \leq i \leq k} A_i \Rightarrow \bigvee_{1 \leq j \leq l} B_j$ is a tautology. As widely adopted, empty conjunctions respectively disjunctions are considered to be *true* respectively *false*. The sequence $A_1, \ldots, A_k$ is often called *antecedent* while the sequence $B_1, \ldots, B_l$ will be named *succedent*. For both, the general term is *cedent*.

4.2 Inference Rules

A proof system allows us to prove that a sequent is a tautology by manipulating its syntax. For this purpose, such a system usually consists of axioms and inference rules. The proof starts at the bottom and new sequents are written on top of the old one. To finish a branch, the axioms are needed to check whether this sequent is an initial sequent.

Definition 6. For a formula $A \in PBF_A$ and arbitrary cedents Γ, Γ', Δ and Δ', $\Gamma, A, \Gamma' \to \Delta, A, \Delta'$ is an *axiom*, also referred to as *initial sequent*.

Lemma 9 (Correctness of Axioms)**.** *All initial sequents are valid.*

Proof. Consider any assignment φ for which the formula F corresponding to the antecedent evaluates to true. As F is a conjunction of its parts, $\varphi(A) = 1$ has to hold. Since the formula G associated with the succedent is a disjunction of its parts, $\varphi(G) = 1$ holds and verifies the correctness of all axioms. □

The inference rules of a sequent calculus explain how sequents can be rewritten or modified. We adapt some of the inference rules from [Bus98] to handle conjunctions and disjunctions. In order to handle compositions, we propose rules to modify

them. To start with, we present the inference rules for conjunction and disjunction. In every rule, the first component in the cedent will be modified or considered but all of them can be used up to commutativity.

Definition 7 (Inference Rules for Conjunction and Disjunction)**.**
Let Γ, Δ be arbitrary cedents and $F_0, \dots F_n \in PBF_A$ some formulas.

$$\wedge\text{: } \textit{left} \quad \frac{F_0, \dots, F_n, \Gamma \to \Delta}{\bigwedge_{1 \le i \le n} F_i, \Gamma \to \Delta}$$

$$\vee\text{: } \textit{left} \quad \frac{F_0, \Gamma \to \Delta \qquad \dots \qquad F_n, \Gamma \to \Delta}{\bigvee_{1 \le i \le n} F_i, \Gamma \to \Delta}$$

$$\wedge\text{: } \textit{right} \quad \frac{\Gamma \to F_0, \Delta \qquad \dots \qquad \Gamma \to F_n, \Delta}{\Gamma \to \bigwedge_{1 \le i \le n} F_i, \Delta}$$

$$\vee\text{: } \textit{right} \quad \frac{\Gamma \to F_0, \dots, F_n, \Delta}{\Gamma \to \bigvee_{1 \le i \le n} F_i, \Delta}$$

So far, this is not new to readers familiar with sequent calculi. Prior to introducing the inference rules for composition, we explain their behaviour with a short example.

Example 6. The goal is to prove $\rho_a \wedge ([(\rho_a \wedge \rho_b) \vee \rho_\varepsilon]; \rho_b) \to \rho_a \wedge \rho_b$ to be a tautology. Initially, some known rules are applied and all different kinds of steps are explained below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\rho_a, ([\rho_{ab} \wedge \rho_b] \vee \rho_b) \to \rho_a}\,(Axiom) \qquad \dfrac{\dfrac{\dfrac{}{\rho_a, \rho_{ab}, \rho_b \to \rho_b}\,(Axiom)}{\rho_a, [\rho_{ab} \wedge \rho_b] \to \rho_b}\,(\wedge\text{: } \textit{left})}{\rho_a, ([\rho_{ab} \wedge \rho_b] \vee \rho_b) \to \rho_b}\,(\vee\text{: } \textit{right}) \qquad \dfrac{}{\rho_a, \rho_b \to \rho_b}\,(Axiom)}{\rho_a, ([\rho_{ab} \wedge \rho_b] \vee \rho_b) \to \rho_a \wedge \rho_b}\,(\wedge\text{: } \textit{right})}{\rho_a, ([\rho_a; \rho_b \wedge \rho_b; \rho_b] \vee \rho_b) \to \rho_a \wedge \rho_b}\,(2\times\ \rho; \rho\text{: } \textit{left})}{\rho_a, ([(\rho_a \wedge \rho_b); \rho_b] \vee \rho_b) \to \rho_a \wedge \rho_b}\,(\wedge; \star\text{: } \textit{left})}{\rho_a, ([(\rho_a \wedge \rho_b); \rho_b] \vee [\rho_\varepsilon; \rho_b]) \to \rho_a \wedge \rho_b}\,(\rho; \rho\text{: } \textit{left})}{\rho_a, ([(\rho_a \wedge \rho_b) \vee \rho_\varepsilon]; \rho_b) \to \rho_a \wedge \rho_b}\,(\vee; \star\text{: } \textit{left})}{\rho_a \wedge ([(\rho_a \wedge \rho_b) \vee \rho_\varepsilon]; \rho_b) \to \rho_a \wedge \rho_b}\,(\wedge\text{: } \textit{left})$$

We do not explain every single step but the different kinds of applied rules. We always read such proof from bottom to top and we explain the rules according to the order of their application.
We first split one formula on the left side into two separate ones due to the inference rule for conjunction $\wedge$: *left*. Second, we use the inference rule for composition $\vee; \star$: *left* in order to split the first formula of a composition by appending the suffix ρ_b to both paths and connect them with $\vee$ again. Third, we compose two boxes to a new one by the NFA A. Next, we split the sequent into two sequents due to the inference rule for conjunction $\wedge$: *right* and have to prove both of them to be valid. Except for the axioms which are marked by the empty bar above, the remaining rules are similar to the former ones.

Definition 8 (Inference Rules for Composition)**.**
Let Γ, Δ be arbitrary cedents, $a, b \in T$ any terminals, $F_1, \dots F_n, G_1, \dots, G_l \in CPBF_A$ some formulas and $\Diamond \in \{\vee, \wedge\}$.

$$\rho;\rho\text{: } \textit{left} \quad \frac{[\rho_{ab}; F_1; \dots; F_n], \Gamma \to \Delta}{[\rho_a; \rho_b; F_1; \dots; F_n], \Gamma \to \Delta}$$

$$\rho;\Diamond\text{: } \textit{left} \quad \frac{\Diamond_{1\leq i\leq l}[\rho_a; G_i; F_1; \dots; F_n], \Gamma \to \Delta}{[\rho_a; (\Diamond_{1\leq i\leq l} G_i); F_1; \dots; F_n], \Gamma \to \Delta}$$

$$\Diamond;\star\text{: } \textit{left} \quad \frac{\Diamond_{1\leq i\leq l}[G_i; F_1; \dots; F_n], \Gamma \to \Delta}{[(\Diamond_{1\leq i\leq l} G_i); F_1; \dots; F_n], \Gamma \to \Delta}$$

$$\rho;\rho\text{: } \textit{right} \quad \frac{\Gamma \to [\rho_{ab}; F_1; \dots; F_n], \Delta}{\Gamma \to [\rho_a; \rho_b; F_1; \dots; F_n], \Delta}$$

$$\rho;\Diamond\text{: } \textit{right} \quad \frac{\Gamma \to \Diamond_{1\leq i\leq l}[\rho_a; G_i; F_1; \dots; F_n], \Delta}{\Gamma \to [\rho_a; (\Diamond_{1\leq i\leq l} G_i); F_1; \dots; F_n], \Delta}$$

$$\Diamond;\star\text{: } \textit{right} \quad \frac{\Gamma \to \Diamond_{1\leq i\leq l}[G_i; F_1; \dots; F_n], \Delta}{\Gamma \to [(\Diamond_{1\leq i\leq l} G_i); F_1; \dots; F_n], \Delta}$$

There are three different kinds of rules as it actually does not matter on which side we apply an inference rule for composition. All of them simulate the characteristics of handling compositions in formulas and therefore these rules are sound. The first rule replicates the behaviour of composing two boxes according to the NFA whereas the second one imitates the process of splitting the second formula and prefix these by the box before. The third rule is to split logical operators in the first formula, append the remainder of the composition to every subformula and recombine them again.

As alluded to before, we could combine the second and the third rule due to the list representation for compositions. We refrained from doing this for the sake of readability and in order to maintain the analogy between these rules and the behaviour of relational compositions for formulas.

In [Bus98], the inference rules for conjunction and disjunction are called strong inference rules in contrast to weak structural rules and the cut rule. Weak structural rules can be used to exchange the order of formulas as well as to duplicate or omit formulas in the cedents. The cut rule is a way to shorten proofs by guessing some additional formula A:

$$\frac{\Gamma \to \Delta, A \qquad A, \Gamma \to \Delta}{\Gamma \to \Delta}$$

For now, the cut rule is omitted but can be introduced easily to shorten proofs without compromising the correctness of the calculus. Because of the definition of axioms as nonempty intersection of antecedent and succedent, there is no point in having weak structural rules. Furthermore, rules to exchange the formulas are pointless as all of the rules can be used up to commutativity.

Remark 3. Prior to showing that the presented calculus is sound and complete, we explore in which way the calculus exploits the absence of negations and implications in our domain. Of course, we did not present any rules to handle these operators, so the calculus is not applicable directly but we could extend the existing rules for this expanded domain. If we consider implications and their associativity, we actually could use every implication to separate the formula into antecedent and succedent and the remaining implications can be handled: $F \Rightarrow G \models\!\models \neg F \vee G$. So it is sufficient to handle negations. If the outermost operator of a formula is a negation, we remove the negation by switching the formula to the other side and omitting the negation:

$$\frac{\Gamma \to \Delta, A}{\neg A, \Gamma \to \Delta} \qquad \text{and} \qquad \frac{A, \Gamma \to \Delta}{\Gamma \to \Delta, \neg A}\,.$$

The correctness of this approach follows directly from the semantic meaning of a sequent and the equality for implications above. In this way, any negation can be handled and the calculus could handle a domain containing implications and negations.

4.3 Soundness and Completeness

Calculi are always the syntactic counterpart to some semantic behaviour one wants to check. The goal is to hand over the process of reasoning to a machine. On the one hand, it remains to show that any sequent which can be proven in the proof system is a tautology, i.e. the proof system is sound. On the other hand, it is preferable that any tautology respectively its corresponding sequent has a valid proof in the proof system, i.e. the proof system is complete. The proofs for these two properties are counterparts in some way. For completeness, the proof is started at the bottom and it is proven by induction on the number of connectives that there is a proof for the remaining sequents, so it is considered from bottom to top. For soundness, the initial sequents at the top are shown to be tautologies and it is observed that applying any rule preserves their property of being tautologies, so the proof tree is considered from top to bottom.

Proposition 1 (Soundness)**.** *The presented proof system is sound, i.e. any formula for which a proof exists is a tautology.*

Proof. As argued before, it suffices to show that (i) any initial sequent is a tautology and (ii) the property of being tautology is maintained by any rule application.

to (i) This was proven in Lemma 9.

to (ii) Firstly, consider the inference rules for conjunction and disjunction. It is straightforward to see that $\wedge$: *left* and $\vee$: *right* do not change the formula associated with the sequents so the property is maintained. Considering $\vee$: *left*, there are n sequents of the form $F_i, \Gamma \to \Delta$ where $1 \leq i \leq n$ for which the corresponding formulas are tautologies. Applying the inference rule yields $\bigvee_{1 \leq i \leq n} F_i, \Gamma \to \Delta$ and thus for any assignment satisfying one of the original formulas, the formula H corresponding to the fresh antecedent

is satisfied as well. Likewise, the unique formula corresponding to all succedents is satisfied by (at least) any assignment satisfying H. The proof for $\wedge$: *right* is analogous und hence both rules perpetuate the property of being tautologies.
Secondly, consider the inference rules for composition which are specific to the relational composition. It is straightforward that all of these rules represent one step backwards in the process of removing compositions. Thus, the property is preserved as well.

□

Similarly to [Bus98], we prove a stronger lemma instead of proving completeness directly.

Lemma 10. *Let $\Gamma \to \Delta$ be a valid sequent with m logical connectives, e.g. $\wedge$ and $\vee$. Furthermore let b_l be the maximal number of terms in a disjunction in Γ, b_r the maximal number of terms in a conjunction in Δ and $b = max(b_l, b_r)$. Then, there is a tree-like proof for $\Gamma \to \Delta$ containing fewer than b^m applications of inference rules for conjunction and disjunction.*

Proof. At first, there will be a case distinction for b. Because of the definition of $CPBF_A$, $b > 1$ if there is a disjunction in the antecedent or a conjunction in the succedent. So for $b = 0$, there are no logical connectives and the sequent is an axiom. Therefore, the proof consists of $0 < 1 = 0^0$ inferences [2] . For $b > 1$, the proof is by structural induction on the number of logical connectives, i.e. m.
In the base case, $m = 0$, the sequent does not contain any logical connectives so that $b = 0$. Since $\Gamma \to \Delta$ is a valid sequence, there is at least one atomic symbol occurring in the antecedent and the succedent so that $\Gamma \to \Delta$ is an initial sequent and $0 < 1 = 0^0 = b^0$ applications of inference rules for conjunction and disjunction are needed.
For the induction step, we only consider the different outermost logical connectives of the formulas in the cedents and disregard compositions as these can be removed by inference rules for composition.

- $\wedge$ in the antecedent or $\vee$ in the succedent:
  We only investigate the first case since the second is handled analogously. Let Γ' be the cedent obtained from Γ by removing $\bigwedge_{1 \leq i \leq n} F_i$, then we can infer $\Gamma \to \Delta$ as follows:

$$\frac{F_0, \ldots, F_n, \Gamma' \to \Delta}{\bigwedge_{1 \leq i \leq n} F_i, \Gamma' \to \Delta}$$

  By the induction hypothesis $F_0, \ldots, F_n, \Gamma' \to \Delta$ is a valid sequent with a proof containing b^{m-1} applications of inference rules for conjunction and disjunction. Since $b^{m-1} + 1 < b^m$, the proof consists of fewer than b^m applications of these inference rules.

[2] To clarify this, we use the appropriate definition: $0^0 = 1$.

- $\wedge$ in the succedent or $\vee$ in the antecedent:
  Like before, we only consider the first case since the second can be handled analogously. Let Δ' be the cedent obtained from Δ by removing $\bigwedge_{1\leq i\leq n} F_i$, then we can infer $\Gamma \to \Delta$ as follows:

$$\frac{\Gamma \to F_0, \Delta' \dots \Gamma \to F_n, \Delta'}{\Gamma \to \bigwedge_{1\leq i\leq n} F_i, \Delta'}$$

  By the induction hypothesis $\forall i$ with $1 \leq i \leq n$ there is proof with b^{m-1} applications of inference rules for conjunction and disjunction. $\Gamma \to F_i, \Delta'$. So these proofs have fewer than $n \cdot b^{m-1}$ applications of these inference rules altogether. Since $n \leq b_l$ and $b_l \leq b$, $n \leq b$ holds and $n \cdot b^{m-1} \leq b \cdot b^{m-1} = b^m$. Thus, fewer than b^m applications of these rules are needed to prove $\Gamma \to \Delta$.

□

Proposition 2 (Completeness)**.**
The presented proof system is complete, i.e. there is a proof for any tautology.

Proof. The result follows from Lemma 10. □

In fact, a proof for any tautology does not only *exist*, but we are able to construct it with the following iterative procedure. For every branch in the proof tree which is no axiom, we apply one of all applicable rules either until it is an axiom or no rule is applicable any more. If the sequent is a tautology, this leads to a proof. If not, we reach a *dead end*, i.e. a sequent which is no axiom and for which no rule is applicable any more, at some point.
Because of the weakening rule, a weak structural rule, and the cut rule in different calculi, there can be sequences of rule applications which do not lead to any result since a dead end might have been caused by the application of such a rule. Then, one has to backtrack and try to use different rules. Therefore, these rules are omitted in the presented calculus. The significant property of strong inference rules in [Bus98] is the fact that one never has to backtrack due to their application. This is also the case for the inference rules for composition and we therefore do distinguish between inference rules for conjunction and disjunction and such rules for composition.
It might seem that our calculus is determistic, but there is some non-determinism in the order of rule applications as multiple rules might be applicable at once. This does not prevent us from finding a proof but the order affects the performance. Distinguishing different precedences of rule applications, we explain two different approaches. The first one can be called *split-last* where every non-splitting inference rule is applied exhaustively on both sides prior to applying any splitting rule. The second approach is called *split-variable* and can be separated into different gradings. In general, splitting before applying non-splitting rules exhaustively can lead to multiple same inferences. If the implication does not hold, there is at least one counterexample witnessing this. We might speed up the process of finding a counterexample with the split-variable approach since some non-splitting rules might not have been applied. If the implication holds, the split-last approach is most efficient since a proof that is constructed that way has the least number of inferences.

5 Lattice of Assignment

In this section, we present a second approach to handle entailment checking. To start with, we assume the instance to be composition-free. We present an algorithm to find candidates for counterexamples which disprove the implication and show some of its properties. Afterwards, we weaken this assumption in order to handle instances with compositions symbolically. For the first part, let $F, G \in PBF_A$ some formulas for which $F \Rightarrow G$ is checked.

Idea Exploiting the monotonicity of positive Boolean formulas PBF_A is the key idea of this approach. The goal is to find all minimal satisfying assignments for F and then check for all of them whether G is satisfied by this assignment. If not, a counterexample is found and therefore the implication does not hold. If G is satisfied by any minimal satisfying assignment, there exists no counterexample and thus the implication holds. Analogously, one can look for all maximal unsatisfying assignments for G and check whether F is not satisfied by those assignments. The main challenge is a clever probing of assignments to find all minimal satisfying assignments for F respectively all maximal unsatisfying assignments for G. To achieve this, the structure of the domain of assignments is important.

Lattice of Assignment We shortly recall the key concept of lattices from [Grä98]. A lattice is a tuple $(D, \leq)$ with a domain D and a partial order $\leq$ where any two elements have a supremum, also called *join*, and an infimum, also called *meet*. As seen before, assignments for F can be seen as subsets of all boxes occurring in F. Given the subset relation, it is straightforward to see that the join is the union of two assignments while the meet is the intersection. Altogether, this yields to the *lattice of assignments* $(GS(F), \subseteq)$ which is shown for a formula with three boxes in Fig. 5.

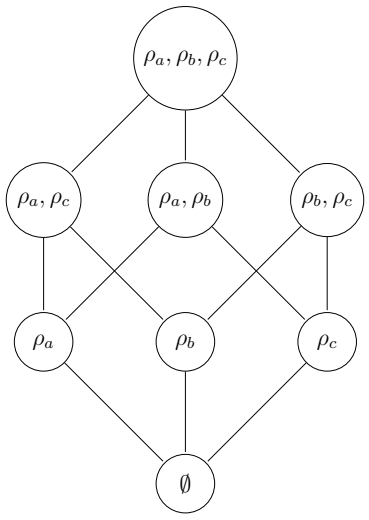


Figure 5: Hasse diagram of a lattice $(GS(F), \subseteq)$ with $|GS(F)| = 3$.

Our algorithm comprises one important optimization which reduces the size of the search space. Therefore, we explain this prior to presenting the algorithm.

Fixing Assignments We identify parts of an assignment that can be fixed without compromising the possibility of finding a counterexample. This is advantageous since any fixed element cuts the size of the lattice of assignments by half. Thus, fixing l components decreases the search space to $\frac{1}{2^l}$.
If $F \Rightarrow G$ holds, *every* assignment $\varphi \subseteq GS(F) \cup GS(G)$ satisfies $\varphi(F) \leq \varphi(G)$. Consider a ground symbol f which only occurs in F and two assignments φ and φ' that only differ in this ground symbol: $\varphi' := \varphi \uplus \{f\} \supseteq \varphi$. We can observe that $\varphi(G) = \varphi'(G)$ as the difference between φ and φ' is invisible for G. Therefore, if φ is a witness for a counterexample, i.e. $\varphi(F) \wedge \neg\varphi(G)$, then φ' is a witness as well because of the monotonicity of F and the fact that $\varphi(G) = \varphi'(G)$. Thus, F can be strengthened by fixing $\varphi(f)$ to *true* for each ground symbol only occurring in F. In a similar way, G can be weakened. Consider a counterexample for which a box only occuring in G exists. F cannot tell the difference between this counterexample and an assignment not containing this box. G is not satisfied by both, so both assignments are counterexamples. Therefore, we assign every box solely occuring in G to *false*.

5.1 Basic Algorithm

Preconditions Even if the goal is to handle compositions, the following example shows why it is reasonable for the formulas to be composition-free for now. In Section 5.2, we will present an algorithm for which this assumption can be weakened.

Example 7. Consider the following instance for which we check entailment:

$$\rho_{ab} \Rightarrow \rho_a; (\rho_b \vee \rho_{ab})$$

The implication does hold since the formula on the right is congruent to $\rho_{ab} \vee \rho_{ab}$ which is logically equivalent to ρ_{ab}, so both sides are actually the same. However, this cannot be handled correctly in terms of assignments. Checking entailment means checking every assignment. The size of an assignment for this instance is intuitively four as there are four different boxes, but actually there is only one single box. On the one hand, it is unclear how to handle compositions of truth values. Composing two truth values instead of the corresponding boxes is impossible since there is not enough information to imitate the behaviour of composing boxes properly, i.e. there is no means of connecting the paths through boxes. On the other hand, one can never be sure that every assignment has been checked since there might be some boxes hidden by a composition.

From now on, we explain the construction and ideas to find minimal satisfying assignments for F. The algorithm to find maximal unsatisfying assignments for G is similar.

Number of Minimal Satisfying Assignments Finding the minimum satisfying assignment φ for F, i.e. $\nexists\varphi', |\varphi'| < |\varphi| : \varphi'(F)$, is NP-*complete* [DG96]. In this use case, one does not want to find *a minimum* satisfying assignment but *all minimal* satisfying assignments φ for F, i.e. $\nexists\varphi' \subset \varphi : \varphi'(F)$. It is easy to find *one* such minimal assignment by reducing a given assignment but the maximal number of such assignments is given by the breadth of the lattice, i.e. the size of the largest antichain. It is straightforward that all vectors of dimension k, where the number of 1's is $\lfloor \frac{k}{2} \rfloor$, are incomparable. Therefore, we count the number of these vectors combinatorially [AZ98]:

$$\binom{k}{\lfloor \frac{k}{2} \rfloor} \overset{(k \geq 2)}{\geq} \frac{2^n}{n}.$$

Thus this is a lower bound on the size of the largest antichain. For an upper bound, the size of the lattice can be considered. Since this is 2^k and thus exponential, the largest antichain has exponential size. So we know that finding all maximal unsatisfying or all minimal satisfying assignments can be costly in the worst case.

Longest Unmarked Path Thinking of the structure of the lattice again and exploiting the monotonicity, the following approach seems to be quite powerful. At the first step, guess an assignment φ in the middle of the lattice, i.e. a subset of size $\lfloor \frac{k}{2} \rfloor$ and check whether F is satisfied. If so, we mark every superset of φ with a label (S) as they are larger than φ and F will be satisfied by them as well. If not, we mark every assignment which is a subset of φ with a label (U) since φ is larger and smaller assignments cannot satisfy F.
Intuitively, we would like to mark as many assignments as possible in one step. Therefore, choose the longest unmarked path in the lattice and jump into one of the assignments bisecting this path. Then, check F and mark accordingly like before. Finding the longest unmarked path is central to this idea. In [BBCB16], a solution to a *similar* problem is presented. There, a SAT-instance is created that describes the marked vertices. With a request of finding a minimal solution, the lower vertex is found. Then, the SAT-Solver is asked to find a comparable vertex which is maximal. Obviously, this approach does not guarantee to find the longest path but its performance seems to be fine overall. Intuitively, including a SAT-Solver which is requested several times per instance produces an immense overhead in our use case. For their problem, this is adequate since the corresponding checking process involves tasks from complexity classes above NP. However, this approach causes too much overhead for our needs.
It could be appropriate to check more assignments to find the minimal satisfying ones as this evaluation is linear. So checking more candidates could improve the overall performance.

Checking Too Much At the beginning, we said that all minimal satisfying assignments have to be checked but we are not restricted to only checking these. So this observation gives rise to the question whether we can exploit that our checking process, i.e. the evaluation of two formulas, has linear complexity. To simplify

the following explanations, we use a notation of vectors for assignments. For this purpose, enumerate all occurring boxes. E.g.

$$\varphi = \{\rho_a, \rho_{ab}\} \subseteq \{\rho_\varepsilon, \rho_a, \rho_b, \rho_{ab}\} \text{ is represented by } (0101)$$

Prior to presenting the basic algorithm, we show the different steps with an example.

Example 8. Consider $F = \rho_a \vee (\rho_{ba} \wedge (\rho_b \vee \rho_{ab}))$, $G = \rho_a \vee \rho_{ba} \vee \rho_\varepsilon$ and the instance $F \Rightarrow G$. Both formulas have four boxes in common. As argued above, ρ_ε can be set to *false* in any assignment which we should check. So the assignments are vectors of size four: $(\rho_a, \rho_b, \rho_{ab}, \rho_{ba})$.

We go through the traversal step by step and explain the different decisions with

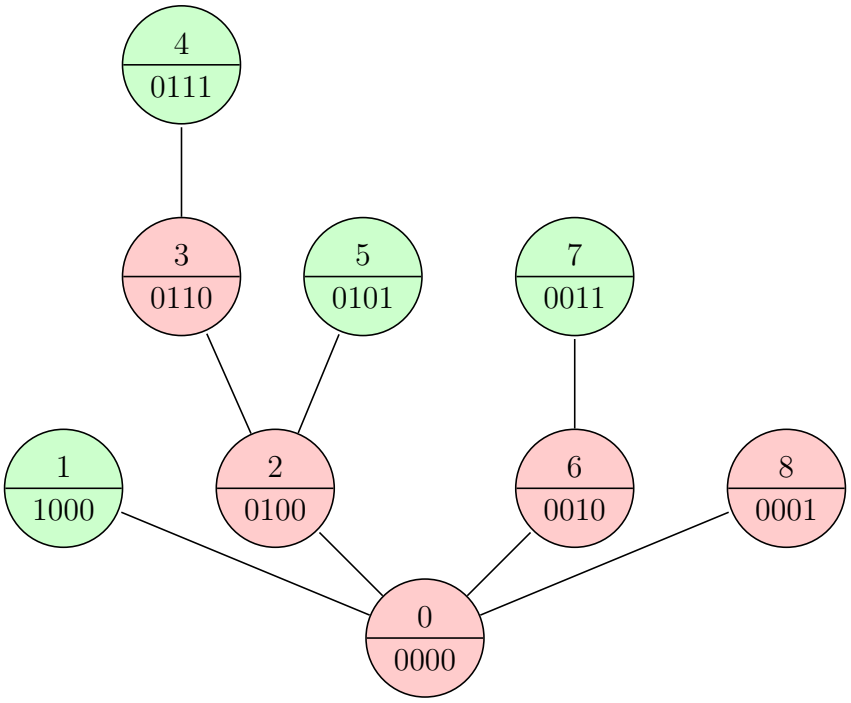


Figure 6: The lower part of the nodes show the considered assignment whereas the upper part gives the order of traversal. Green nodes contain satisfying assignments and red nodes contain unsatisfying ones.

Figure 6. The empty assignment is also checked as F might be satisfied by the fixed assignment for the boxes which only occur in F. Here, this is not the case and hence the node is red. So we proceed with $\varphi_1 = (1000)$ and evaluate F. Since $\varphi_1(F) = \mathit{true}$, we found a possible candidate for a counterexample and evaluate G. $\varphi_1(G) = \mathit{false}$ and thus φ_1 is no witness of a counterexample and in general there are no counterexamples since the implication does hold. So from now on, we only point to the assignments we would check G for. Since we backtrack once, we never assign ρ_a to true again and continue with the assignments $\varphi_2 = (0100)$ and $\varphi_3 = (0110)$. As $\varphi_2(F) = \mathit{false}$ and $\varphi_3(F) = \mathit{false}$, we continue with $\varphi_4 = (0111)$ and check G as $\varphi_4(F) = \mathit{true}$. Even if we had not found a satisfying assignment, we would have proceeded by backtracking as there are no more options in this branch.

The backtracking leads to node ② with its remaining child $\varphi_5 = (0101)$. Like before, we check G since $\varphi_5(F) = \mathit{true}$. Backtracking leads back to the root ⓪ this time and we go on with $\varphi_6 = (0010)$ and $\varphi_7 = (0011)$ similarly. Here, we see the scenario described above: Even if $\varphi_7 \subseteq \varphi_4$, we checked both assignments. Finally, assignment $\varphi_8 = (0001)$ is considered in the same manner and we complete our example.

After the explanation of the algorithm on the basis of the example above, we provide some sample code for the required methods and give some more insights afterwards.

```
1  bool checkWithMinSatAssignments (Formula F, Formula G) {
2    // the assignment for common boxes
3    vector<int, bool> a = {false, ..., false};
4    for (int r = 0; r < a.size(); r++) {
5      //try to find a counterexample at this level
6      if (!checkPerLevel(F, G, a, r)) {
7        return false;
8      }
9    }
10   return true;
11 }
12 bool checkPerLevel (Formula F, Formula G, vector a, int i) {
13   a[i] = true;
14   // check current assignment
15   if (evaluate(F, a)) {
16     if (!evaluate(G, a)) {
17         return false;    // counterexample
18     }
19     else{
20         return true;     // reset (MSA was found) and go on
21     }
22   }
23   else { //keep looking for MSA
24     i++;
25     for (int r = i; r < a.size(); r++) {
26       if (!checkPerLevel(F, G, a, r)){
27         return false;    // counterexample
28       }
29     }
30     return true;         // reset and go on
31   }
32 }
```

To start with, we should mention that the formulas F and G are simplified due to the fixed assignment for boxes occuring in exactly one formula. After this simplification, we catch the trivial case in which F is already satisfied by the fixed part as the implication does not hold in this case.
Then, the first method only initializes the process in line 3 and calls the second one

for any index from which the traversal should be started with the for-loop in line 4, e.g. it spawns the branches starting in the root. At first, the second method sets the current index to *true* in line 13 and checks whether formula F is satisfied. If so, G is checked and depending on the result the algorithm returns *false* (line 17) if a counterexample is found, i.e. G is not satisfied, and *true* (line 20) if G is satisfied, as a minimal satisfying assignment for F is found. If F is not satisfied, we keep looking for a minimal satisfying assignment and thus iterate from this index with the for-loop in line 25. There, the method is called recursively with a larger index.

The Other Way Around As alluded to before, we might not only check the minimal satisfying assignments because of the traversal. Considering Example 8 and the nodes ④ and ⑦ with their assignments 0111 and 0011 again, such scenarios arise if a variable on the left of an assignment could be set to *false* without altering the overall evaluation. If we traverse the lattice the other way around after the initial check, i.e. we start iterating with the last box and continue rightwards every time, the order of the traversal changes to ⓪, ⑧, ⑥, ⑦, ②, ⑤, ③, ④, ①. Thus, we could collect all minimal satisfying assignments and check whether a smaller assignment has been checked before we evaluate the current one. As all minimal satisfying assignments are unrelated, this check is linear in the length of the set as we have to compare the new assignment with any element in the set in the worst case. This happens if we find a minimal satisfying assignment that is unrelated to each of those seen before. A single check depends on the number of boxes in the assignment. On the one hand, such an approach will only improve the overall performance if the formulas become very large in relation to the number of boxes as we could avoid expensive evaluations thereby. On the other hand, such formulas are likely to be reducible and, as we argue in Section 9.2, this is more appropriate in the context of an iteration process.

Ordering Boxes In the last paragraph, we explained that it might be useful to traverse the lattice in a slightly different way. Considering the order of boxes in an assignment, we could count their occurence in one or both formulas and reorder these for the traversal. At this point, one can think of different heuristics. On the one hand, we could put the boxes in front which occur in F very often, as the assignment consisting of these might be a small one satisfying F. In addition, we can include that the boxes are not allowed to occur in G very frequently as the satisfying assignments for F might satisfy G as well. This is undesirable as these assignments are no counterexamples in this case. On the other hand, the order could be determined by the minimal number of occurrences in G. As pointed to before, it is important to find a good trade-off between the overhead to enable such heuristics and the benefits to enhance the overall performance.

Reusing Evaluation Results We can exploit the similarity between two succeeding assignments in the sequence of evaluations of the considered formula. If we do not backtrack, we ascend in the tree which is created during the traversal of the lattice and the assignment only changes at one position. Hence, it is useful to save the

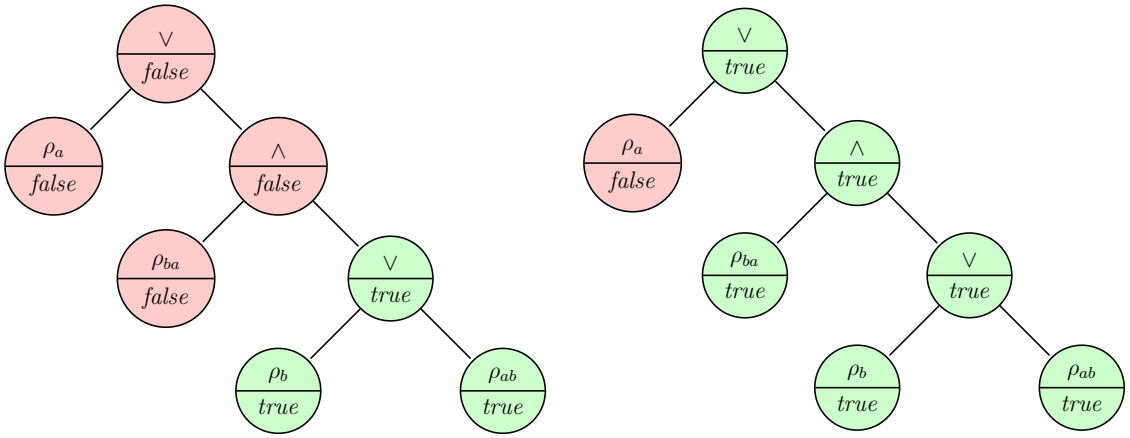


Figure 7: Indexed syntax tree for $F = \rho_a \vee (\rho_{ba} \wedge (\rho_b \vee \rho_{ab}))$ for the two assignments 0110 and 0111 which differ in exactly one box.

intermediate results of the previous step and only recompute the changed part. To illustrate one possible technique, we can think of the syntax tree of a positive Boolean formula. The inner nodes represent logical operators whereas the leaves represent boxes. For the formula $F = \rho_a \vee (\rho_{ba} \wedge (\rho_b \vee \rho_{ab}))$ in Example 8, such a syntax tree is illustrated in Figure 7. We can reuse the intermediate results for the assignment 0110 to evaluate F for 0111. Therefore, we jump to any occurence of the box whose assignment changed, ascend the syntax tree and check whether the intermediate result of the inner node changes. If not, we stop as the overall evaluation result does not change. If so, we continue and look at the parent of the current node. By this, we reach the root iff the evaluation result changes due to the minimal change in the assignment. E.g. we change ρ_{ba} to *true* in Figure 7 and propagate this change upwards. As the intermediate results of all inner nodes change, we reach the root and the result of the evaluation becomes *true*.
To apply this technique for the case of backtracking, we either change multiple parts of the assignment, e.g. use the intermediate results of ④ for ⑤, or use the immediate predecessor in the computation tree, e.g. use the intermediate results of ② for ⑤.

Remark 4. Before we show how compositions can be handled symbolically, we explore in which way this approach depends on the absence of negations and implications. In contrast to the sequent calculus, we cannot extend this algorithm easily to this expanded domain. The reason is the key idea of the presented algorithm which is to find candidates for counterexamples cleverly and if none exist, the implication has to hold since any valid candidate was checked. The actual problem is that we cannot look for these candidates cleverly as any assignment is such a candidate. Due to the loss of monotonicity in the expanded domain, we are not able to cut branches and find promising candidates which is central to the presented algorithm. So overall, this algorithm strongly depends on the fact that only positive Boolean formulas can occur.

In the next section, we investigate an approach where both formulas can contain compositions not preventing us from finding real counterexamples.

5.2 Refining Compositions

From now on, let $F, G \in CPBF_A$ be some formulas with compositions. As argued before, there is no reasonable way to evaluate two composed boxes with their truth values, thus one appropriate solution to include compositions is to handle them as ground symbols. For this section, the set of boxes $B(F)$ does not coincide with the ground symbols $GS(F)$. Hence, we have defined an assignment as a subset of ground symbols and not as a subset of boxes. With the compositions $C(F)$ in F, the ground symbols are the disjoint union of boxes and compositions in F: $GS(F) = B(F) \dot\cup C(F)$.

Real vs. Spurious Counterexamples In order to handle compositions, we will refine them until F and G do not share any of them and fix their assignment appropriately. As explained before, we look for *real* counterexamples, in contrast to *spurious* ones. Intuitively, a *real* counterexample is an assignment for which the compositions cannot be refined in a way that a counterexample becomes invalid, e.g. if new boxes due to the refining are considered. Technically, φ is a *real* counterexample if $\varphi \dot\cup X$ is a counterexample where X is a subset of the new boxes which evolves from refining the compositions. A counterexample is *spurious* if it is not real. Basically, this can be checked by removing all compositions and checking for every assignment of the new boxes whether one of them is a counterexample. If not, the counterexample is spurious. E.g. consider that F and G have one composition in common which is assigned to *false* for some witness φ of a counterexample. So the trustworthiness of this witness has to be examined. The question whether φ weakens G even if a refined version is valid cannot be answered in general. The most appropriate way to handle these scenarios is to refine all compositions until F and G do not have any of them in common. Then, we set all compositions in F to *false* in order to weaken F and all compositions in G to *true*. Thereby, a counterexample cannot be spurious as we can only strengthen F and weaken G if we refine compositions and this does not impair a counterexample.

Adopting the Basic Algorithm As the assignment for compositions is fixed, we can easily derive a new algorithm using the basic one. Here, two auxiliary methods are used which return a Boolean whether F and G have a composition in common or whether there are remaining compositions in a formula, respectively. In order to explain the whole process, we refer to the method on the right.
Initially, with the while-loop in line 2, the precondition that both formulas do not share any compositions is ensured. Then, the known algorithm for checking the formulas with minimal satisfying assignments is called in a while-loop (line 5). As long as no counterexample is found, we check whether there are compositions to refine in F or G in line 6. If not, we return *true* since every candidate for a counterexample was checked in line 5 before. Otherwise, we refine the compositions in F and G. Like before, the while-loop in line 10 ensures that both formulas do not share any compositions. If we find a counterexample during the while-loop in line 5, *false* is returned.

```
1  bool checkRefiningCompositions (Formula F, Formula G) {
2    while(compositionsInCommon(F, G)) {
3      refine(F, G);
4    }
5    while(checkMinimalSatisfyingAssignments (F, G)) {
6      if(noCompositions(F) && noCompositions(G)) {
7        return true;
8      }
9      else {
10       while(compositionsInCommon(F, G)) {
11         refine(F, G);
12       }
13     }
14   }
15   return false;
16 }
```

Refining Compositions Since we did not specify how compositions are refined, the given program fragment is a template where the process of refining can be specified in different ways. This method has two parameters since there might be heuristics handling both formulas in parallel for example.
One can think of a version in which the method actually does not only refine the compositions but removes them completely. In this way, there will be two iteration steps at most. At first, there is one check where the compositions are handled symbolically if both formulas do not share compositions. In the second round, both formulas are composition-free and checked again. This approach can be understood as a first quick check for some simple cases followed by a larger one if necessary.
Another extreme is a version where compositions are refined very gradually. Pictorially, compositions are pushed one level deeper in the syntax tree of the formula in order to uncover the logical operators underneath. Of course, this approach can need a lot more iteration steps. If all compositions have to be refined, the last check coincides with the second one in the first version which is the worst-case scenario for this approach. It is worth mentioning that this procedure primarily speeds up the process of finding a counterexample. If the implication does hold, it is likely that most compositions have to be removed.

Even More Possibilities It could even be possible to connect the idea of the basic algorithm with the process of refining. Consider a run where no counterexample have been found and thus (at least) all minimal satisfying assignments M for F have been computed. Then, we could solely refine the compositions in G and just recheck for any assignment in M. Fulfilling the condition that no composition occurs in both formulas leads to a great disadvantage. It may be likely that storing M is not only useless since it cannot be used but even very space demanding considering the maximal number of unrelated assignments which is exponential.

6 Exploiting Iterations

Until now, we have solely considered two arbitrary formulas and ignored that the formulas to check are constructed by the same function, i.e. by the same equation. The next two sections try to exploit the fact that two formulas are built iteratively and thus similarly. At first, we show how to reduce an instance by omitting suitable parts that did not change after the last update. Second, we present an attempt how one could split compositions and check them seperately and proceed with a counterexample which shows that the approach actually does not work.

6.1 Checking Changed Parts

In general, only formulas for the same non-terminal X are compared. So after a few iterations the outermost operator stays the same since the ownership of X does not change. There may be parts of the new formula which already existed in the former one since not every component of the right-hand side has changed compulsorily. Actually, we have to compute again if at least one component of the right-hand side has changed. So exploiting these observations gives rise to the following two lemmata. For the remainder of this subsection, let $F_1, \ldots, F_n, F'_1, \ldots F'_n \in CPBF_A/_{\Leftrightarrow}$ for some $n \in \mathbb{N}$, where F_i and F'_i represent the formula built by the same part of the equation.

Lemma 11. *Let $G = F_1 \wedge \cdots \wedge F_n$ and $G' = F'_1 \wedge \cdots \wedge F'_n$. Furthermore, $G \Rightarrow G'$ as G' is assumed to be the successor of G in the Kleene iteration. Let I be the index set of changed components in G and G', i.e. $\forall i \notin I : F'_i = F_i$.*
Then $G' \Rightarrow G$ iff $F'_1 \wedge \cdots \wedge F'_n \Rightarrow \bigwedge_{i \in I} F_i$.

Proof. The direction from left to right trivially holds since the $G \Rightarrow \bigwedge_{i \in I} F_i$ and the implication is transitive.
Vice versa, we use the sequent calculus presented in Chapter 4 to prove the result. So let us have a look at a part of the proof tree of $G' \Rightarrow G$:

$$\dfrac{\dfrac{F'_1, \ldots, F'_n \to F_1 \qquad \ldots \qquad F'_1, \ldots, F'_n \to F_n}{F'_1, \ldots, F'_n, \to F_1 \wedge \cdots \wedge F_n}\ (\wedge\text{: } right)}{F'_1 \wedge \cdots \wedge F'_n \to F_1 \wedge \cdots \wedge F_n}\ (\wedge\text{: } left)$$

So it suffices to prove $F'_1, \ldots, F'_n \to F_j$ for $1 \leq j \leq n$. For $k \notin I$, the corresponding sequent is

$$F'_1, \ldots, F'_{k-1}, F_k, F'_{k+1}, \ldots, F'_n \to F_k$$

and they are all axioms due to the definition of initial sequents and hence valid sequents. Consequently, proving $F'_1, \ldots, F'_n \to F_i$ for $i \in I$ is sufficient. Recombining the remaining sequents, i.e. applying the rules of the sequent calculus backwards, yields the desired result. □

The next lemma is the counterpart to the latter one and shows how to deal with $\vee$ as outermost operator.

Lemma 12. *Let $G = F_1 \vee \cdots \vee F_n$ and $G' = F'_1 \vee \cdots \vee F'_n$. Furthermore $G \Rightarrow G'$ as G' is assumed to be the successor of G in the Kleene iteration. Let I be the index set of changed components in G and G', i.e. $\forall i \notin I : F'_i = F_i$.*
Then $G' \Rightarrow G$ iff $\bigvee_{i \in I} F'_i \Rightarrow F_1 \vee \cdots \vee F_n$.

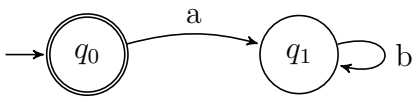
Proof. The proof is analogous to the one for Lemma 11. Instead of using $\wedge$: *left* and $\wedge$: *right*, $\vee$: *right* and $\vee$: *left* are applied. The remaining reasoning is analogous. □

6.2 Splitting Compositions

In Chapter 7, we argue why a normal form for the grammar similar to the Chomsky Normal Form is reasonable. Therefore, we try to exploit this observation and will see that this does not work out.

Attempt to reverse Lemma 3 In Chapter 2, we stated Lemma 3: If $F \Rightarrow F'$ and $G \Rightarrow G'$, then $F; G \Rightarrow F'; G'$. The question might arise whether the inversion holds as well. It is easy to see that this is not the case in general. So we can try to find assumptions satisfiable by the iteration to exploit the normal form. Therefore, we could try to prove the following statement:
$G \Rightarrow G'$ *iff* $F; G \Rightarrow F; G'$ or the pendant where the second component is unchanged. The benefit of such a lemma is obvious as we could abbreviate the process of checking implication and only have to check it if both components of the composition changed. We were able to find a counterexample which is presented in the following.

Example 9. We consider the context-free grammar and the automaton given by Figure 8. On the right, the Kleene iteration is shown where all formulas are updated at once. Here, we see that $\rho_a; (\rho_\varepsilon \vee \rho_b) \Rightarrow \rho_a; \rho_\varepsilon$ but $\rho_\varepsilon \vee \rho_b$ does not imply ρ_ε and observe that the desired statement is violated.



$G = (\{A_\square, B_\bigcirc, C_\bigcirc\}, \{a, b\},$
$\{A \to \varepsilon \mid C,\ B \to a; A,\ C \to b\})$

$A_\square$	$B_\bigcirc$	$C_\bigcirc$
false	*false*	*false*
ρ_ε	*false*	ρ_b
$\rho_\varepsilon \vee \rho_b$	$\rho_a; \rho_\varepsilon$	ρ_b
$\rho_\varepsilon \vee \rho_b$	$\rho_a; (\rho_\varepsilon \vee \rho_b) \cong \rho_a \vee \rho_a$ $\bowtie \rho_a = \rho_a; \rho_\varepsilon$	ρ_b

Figure 8: Example consisting of an automaton, a context-free grammar in which symbols indicating the ownership are omitted in the production rules, and the corresponding Kleene iteration.

7 Implementation

At first, we explain the structure of our tool and its components. Then, we present how we implemented the theory about formulas without compositions and point to the challenges for versions with compositions.

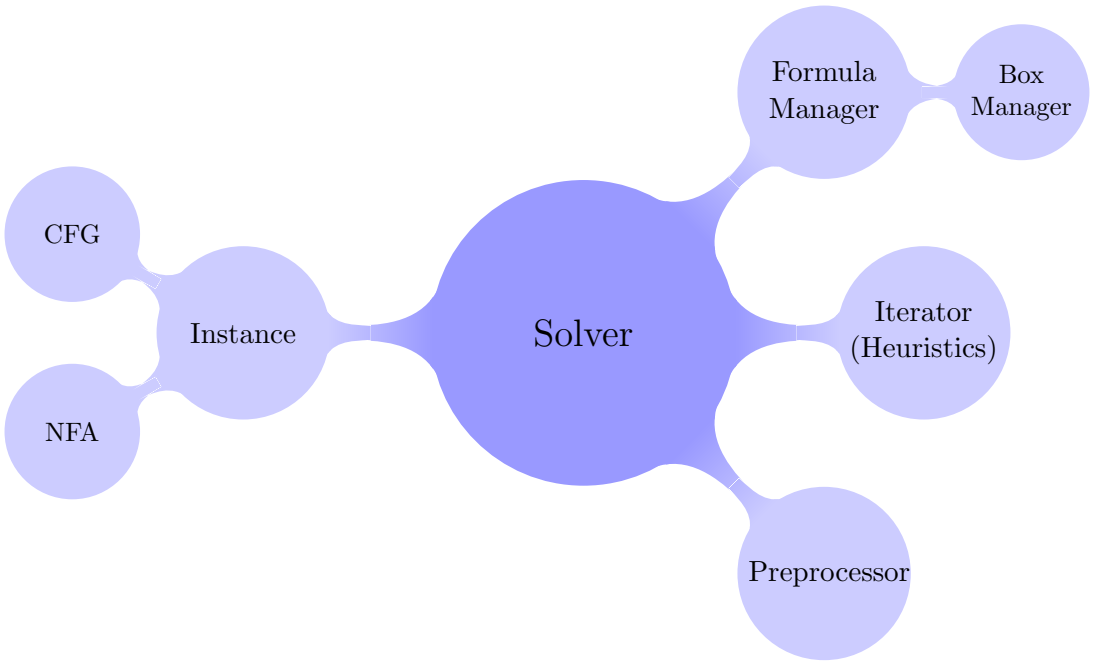


Figure 9: The input of the solver consists of an NFA and a CFG which is preprocessed at first. The iterator comprises heuristics and tells the solver which non-terminal to update next and whether to check implication for example. The formula manager handles anything related to formulas like implication checks.

7.1 Structure of the Tool

We have developed a tool to solve context-free games with the summary technique from [HMM16] and explain the structure and the different components as shown in Figure 9.

Random Generator

Guided by an execution of the tool, we start with the parameters of the random generator which produces the instance consisting of the automaton and the context-free grammar. For both, the size of the alphabet can be set.
For the automaton, we can adjust the number of states and the probability whether there is a transition from one state to another labelled by one terminal. Furthermore, we can force the automaton to be connected so that any state is reachable from the initial one. By default, there is only one initial state, but we can fix a probability

with which any other state is initial as well, so there is always at least one initial state. Similarly, there is a probability for final states but there might be no final state in contrast.
Considering the context-free grammar, we can set the number of non-terminals and in order to have balanced games, the probability that a non-terminal is owned by refuter is one half. We assume a normal form which is similar to the Chomsky Normal Form and briefly sketch how to model such a grammar.

Modelling the Grammar Context-free grammars are a good approximation to model the control flow of a programming language. From a sequential point of view, it is appropriate to think of the execution of the code line by line. Consider n lines of code labelled with $l_1, \ldots, l_n$ without any branching or loops. We can state the following rules with initial symbol S:

$$S \to l_1 R_2, R_2 \to l_2 R_3, \ldots, R_i \to l_i R_{i+1}, \ldots, R_n \to l_n$$

So we are able to model the behaviour of sequential code and we therefore can represent its behaviour by one single non-terminal. Moreover, this kind of rules can be transformed into Chomsky Normal Form [HU79] by introducing new non-terminals for terminal symbols and large right sides.
How to model $if - then - else$ was shown in the introduction when we explained the difference between demonic and angelic non-determinism. If we consider angelic non-determinism, C is owned by prover and the corresponding rule is $C \to T \mid E$ where T is the non-terminal representing the branch where the condition holds and E the one where it does not hold. In case of demonic non-determinism, C is owned by refuter and the condition *cond* is comprised as follows:

$$C \to assume(cond).T \mid assume(\neg cond).E.$$

It remains to show how we model method calls. Therefore, we consider this short example code where C is the non-terminal where the subroutine is called, R corresponds to the code after the subroutine has been invoked and S corresponds to the code of the subroutine.

```
C: subroutine();          S: subroutine() {
R: ...                        ...
                            }
```

The goal is to pass the return-address R of any possible caller to the callee so that the control flow can proceed after the subroutine. This behaviour can be modelled by the following rule: $C \to SR$.
In contrast to Chomsky Normal Form, we allow ε-rules, i.e. rules where ε (the empty word) is on the right-hand side, in order to enable a change of control from one player to its opponent. So overall, the control flow can be modelled by a normal form similar to Chomsky Normal Form with rules of the form:

$$A \to BC \mid a \mid \varepsilon$$

where A, B and C are non-terminals and a is a terminal.

Preprocessor

After the instance was created, we first minimize the grammar because of the following observation. As we start the fixed-point iteration with *false*, the formula for a non-terminal never changes to anything else if it is a conjunction and it occurs on the right side of its own production rule or any of its parts stays *false* constantly. Moreover, the formula for a non-terminal never changes if all components on the right side of its production rule stay *false* all the time, regardless of the outermost operator. The set of those non-terminals can be computed with a fixed-point iteration. We start with an empty set $\mathcal{S}$ of non-terminals. In every round, we add any non-terminal $X \in N$ to $\mathcal{S}$ for which one of the following conditions holds for the corresponding equation until we reach a fixed point. The resulting formula

- is a conjunction and at least one of the non-terminals on its right side is in $\mathcal{S} \cup \{X\}$.
- is a disjunction and there is at least one non-terminal of $\mathcal{S} \cup \{X\}$ in any co-clause.

This procedure terminates after at most $|N|$ steps. Since $\mathcal{S}$ is the set of non-terminals for which the corresponding formulas stay *false* all the time we omit these in the grammar.

Solver, Iterator and Formula Manager

Then, the solver starts the fixed-point iteration to compute the formulas of the remaining non-terminals. For this purpose, the solver asks the iterator which non-terminal to update next. We do not update the formulas for all non-terminals at once but each of them seperately, so that recent changes can be exploited. The iterator can use a fixed order for instance where the order of updates does not depend on recent changes but is the same all the time. In contrast, we can use a worklist and only update non-terminals for which at least one component on its right side has changed.
Then, the solver incorporates the formula manager to combine the different parts for the new formula. If requested, it checks whether the new formula implies the old one. Whenever the formula manager needs to compose boxes it calls the box manager that is the interface to the automaton and thus handles the boxes. The formula manager is responsible for anything concerned with formulas and therefore the part we consider for the remainder of this section.

7.2 Formulas without Compositions

In this section, we explain the implementation of the approaches where compositions are removed to check implication.

Formulas in CNF

In Chapter 3, we presented three polynomial algorithms for normalized formulas. The implementation of the first one for which both formulas are given in CNF is quite straightforward. First, we remove all compositions and transform the result into CNF afterwards. Checking for any clause in F that there is a clause in G that is superset of the first one is not difficult neither. We could also have chosen the algorithm for which both formulas have to be given in DNF. The polynomial algorithm for which the formulas are given in different normal forms would be disadvantageous as we usually check implication after each step of the fixed-point iteration. To be specific, we would have to transform a formula from DNF to CNF as it was used in CNF before and has to be in DNF for the next check. These transformations can be exponential [MRW05] and are thus not preferable.

Lattice of Assignment

For the algorithm in Chapter 5, we only remove compositions and the formulas stay non-normalized. In our implementation, the results satisfy the alternating operators assumption, i.e. that any component of a conjunction is a disjunction and vice versa, because of the known advantages. By the usage of caches for the removal, we abbreviate and speed up this process. To use them properly, we would like to identify syntactically equivalent formulas easily. We can enforce this uniqueness by some inductive reasoning. First, we have exactly one formula for every box. For every conjunction or disjunction of smaller formulas, we enforce uniqueness by checking whether this formula was built before. Therefore, we use caches for conjunctions and disjunctions as well. Then, the presented methods can be employed straightforward. The performance of these procedures is crucial for the overall performance as they are invoked very frequently.

Iterative Backtracking Thus, the recursion which is responsible for the backtracking is substituted by an iterative backtracking procedure. This method is used similarly in SAT-Solvers and is modified due to the different termination criteria in our use case. The key is the use of a stack and a container of unused variables to iterate over all assignments in the same way but without recursion.

Reusing Evaluation Results Our procedure also incorporates the technique of reusing evaluation results. Consider two formulas F and G for which we check implication with the approach using minimals satisfying assignments for F. Therefore, we construct an auxiliary data structure for F which serves as current state of evaluation in the beginning. After the initial assignment was set, F is evaluated once exhaustively. In this process, we do not only compute the result of the evaluation

but count the number of satisfied subformulas for every inner node in the syntax tree. This additional information can be used to propagate changing intermediate results faster, as we only increment or decrement this number for every parent and check whether their evaluation result changed with one equality check. The process to ascend the new information is executed every time the assignment of a box is changed so that we can evaluate F simply by checking its result in the auxiliary data structure.
Generally speaking, we changed the complexity of two different methods considering assignments and evaluation. At first, the process of setting an assignment was in $O(1)$ whereas the evaluation was linear. We have observed that there are mostly minor changes in the assignment between two succeeding evaluations for F so we added a method to set one box to another truth value so that the information is propagated directly. For some deformed formulas, the complexity of this method is linear but in general, the ascending paths can be cut very early because of our set representation and the alternating operators assumption. In turn, evaluation of F is in $O(1)$.
As G is not evaluated as often as F we refrain from constructing the whole auxiliary data structure and we only set the truth values of its boxes so that we can evaluate it for the current assignment. In doing so, we exploit intermediate results of F if they share common subformulas, i.e. we check whether we know an intermediate result due to F prior to going into deeper levels of the formula during the evaluation of G.

Minimal Satisfying Assignments vs. Maximal Unsatisfying Assignments So far, we only explained all constructions for minimal satisfying assignments. The procedure is analogous for maximal unsatisfying assignments but we would like to reason why we use both of them. As argued before, every fixed part of the assignment cuts the search space in half. Therefore, we look at the first level in the syntax tree and figure out boxes which can be set to *false* respectively *true* without compromising possible counterexamples. If we try to find minimal satisfying assignments for conjunctions, any box on the first level can be set to *true* as any assignment that does not include this box cannot be a satisfying one. Analogously, if we look for maximal unsatisfying in disjunctions, any box on the first level can be set to *false* as any assignment that does include this box cannot be an unsatisfying one. Overall, we use the approach using minimal satisfying assignments if F is a conjunction and the one using maximal unsatisfying assignments if G is a disjunction. They are both correct for conjunctions and disjunctions but the performance decreases. In case both conditions hold, we invoke the approach using maximal unsatisfying assignments as we know from the iteration process that G is the older formula and thus smaller.

SAT-Solver

For an instance $F \Rightarrow G$ for $F, G \in PBF_A$, we use the Tseytin transformation [Tse70] to get a formula in CNF which is satisfiable iff the original one is satisfiable in polynomial time. More precisely, we know that for $F \Rightarrow G$ the implication holds iff $\neg F \vee G$ is a tautology. Furthermore, a formula H is a tautology iff its negation $\neg H$ is unsatisfiable. Hence, we transform $\neg(\neg F \vee G) \eqdashv F \wedge \neg G$ into a formula N in CNF using the Tseytin transformation. Then, we check whether N is satisfiable with a SAT-Solver. As explained before, N is unsatisfiable iff the implication holds.

7.3 Symbolic Formulas

We have presented two different approaches to handle formulas with compositions in implication checks and will point to the challenges coming along with the implementation of these approaches.

Both Approaches Revisited

Sequent Calculus In the sequent calculus, we might not have to remove all compositions if we can identify the same formula including compositions in the antecedent and the succedent. In order to catch these scenarios, we would like to identify identical formulas easily. We called this uniqueness before and it is even crucial for an actual implementation if we think of deciding whether the succedent and the antecedent have some formula in common, which is used very frequently. Therefore, we have to compare each formula on the left and each on the right pairwise either until we find an overlap or every possible combination has been considered.

Refining Compositions We introduced a refining procedure to handle the formulas symbolically for the lattice of assignment approach. As argued before, we need to ensure that both formulas do not have compositions in common. Imagine the scenario if we do not recognize two identical compositions. Then, we set it to *false* on the left side whereas we set it to *true* on the right side. Consider the trivial example where both formulas only consist of this composition. There, we find a counterexample and have to refine the composition even if it is the identical. In general, every time a composition changes the satisfiability in a significant way and we are not able to identify them being identical, we compute the whole process of finding minimal satisfying or maximal unsatisfying assignments even if we could have refined them right away.

Consequences of Altering Formulas As explained before, the uniqueness of formulas is not essential but it is preferable for the usage of caches to remove the compositions in this case as well. We also ensure uniqueness by the usage of caches, but the fundamental problem in this case is the caching of altering formulas. Basically, the hash-value changes if the corresponding formula is modified so that the normal usage of caches does not ensure uniqueness anymore. Furthermore, our alternating operators assumption could be invalidated, e.g. a subformula of a

conjunction can also become a conjunction if we remove compositions. The latter does not violate the correctness of both approaches but can worsen their performance as logically equivalent formulas may not be identified as such.

Merging Congruence Classes From a theoretical point of view, the goal is to merge different congruence classes efficiently when we update formulas and get the current representative for this class. This ought to be the formula for which the least number of removal steps have to be applied in case of compositions. We could extend this behaviour to the implication checks as well. When we figure out that a newly created formula implies its predecessor, we know that they are logically equivalent. Therefore, we update the new formula to its predecessor as the size of the latter is smaller.

8 Evaluation

In this chapter, we give some insights into the experimental results of our tool. We start with some case studies about the benefit of preprocessing, i.e. the reduction of the grammar, the preassignment of Chapter 7 and the omission of unchanged parts of Chapter 6. Then, we proceed with an extensive comparison of three different approaches to check implication: a polynomial algorithm for formulas in CNF, the approach considering the lattice of assignment and a method incorporating a SAT-Solver.

8.1 Case Studies about Preprocessing

We have presented different preprocessing steps in previous chapters. For each of those, we analyze the benefits for the different approaches. We ran some experiments for different numbers of states in the NFA, terminals and non-terminals in the CFG that are similar to the setting for the extensive comparison which we explain in the next section.

Reduction of the Grammar To start with, we consider the reduction of the grammar. For very small instances, the reduction catches some trivial cases and therefore increases the performance strongly. The improvement for bigger instances was at a low percentage, mostly from one to five percent with rare outliers to 30 %, so this observation does not carry over to bigger instances. This might seem discouraging at first, but it gives more reliance to our benchmarking results because big instances are not reduced to trivial ones. This is positive as it probably would also not happen for real examples.

Preassignment As shown in Chapter 7, we can fix some parts of the assignment without compromising the possibility of finding a counterexample. We have realised that this technique is crucial for the performance of the lattice of assignment approach during our experiments. So we have adopted it to the SAT-Solver approach but the impact was negligible and the performance even suffered slightly at times. Due to the Tseytin transformation we introduce new variables to represent subformulas so that the impact of fixing a small number of them might be marginal. This is one way to explain the stagnation of performance. Moreover, SAT-Solvers use heuristics to speed up the process of finding a satisfying assignment. We surmise that these heuristics are impaired by the preassignment and therefore not advantageous for this approach.

Checking Changed Parts As explained in Chapter 6, we can omit unchanged subformulas on the right side of conjunctions and on the left side of disjunctions if both formulas share the same operator. In contrast to the reduction of the grammar, this is a preprocessing step that is performed prior to each implication check. Both approaches for non-normalized formulas could benefit from this observation.
Prior to the usage of intermediate evaluation results for the lattice of assignment

approach, this technique improved the performance up to ten percent, but there are no differences in our recent experimental results anymore. In order to explain this behaviour, consider two conjunctions F and G for which we check entailment: $F \Rightarrow G$. As argued before, we look for minimal satisfying assignments of F and omit unchanged parts in G. By this means, the evaluation results for any subformula of F are maintained and thus for all unchanged parts as well. As we use these for the evaluation of G, there is no point in omitting these parts anymore because their branches are cut immediately during the evaluation of G.
For the sake of completeness, we also applied this technique to the SAT-Solver approach. As the technique did not result in any impact of the performance either, we assume that the overhead of computing the unchanged parts annihilates its potential to improve the overall performance.

8.2 CNF, Lattice of Assignment and SAT-Solver

According to these observations, we take the best version of every approach for our extensive comparison. So the lattice of assignment approach uses the preassignment whereas the SAT-Solver does not and we do not omit unchanged parts for both. The number of states of the NFA, the size of the alphabet and the number of non-terminals are the parameters for our experiments and we ordered them accordingly in the first column of the table in Figure 10.

Experimental Setting Altogether, the random generator created 100 instances for every configuration and we forced the grammar to be connected. Prior to invoking the different solvers with a worklist iterator, we reduce the instance due to the grammar reduction. The cutoff is 100 seconds so that a solver was interrupted if it exceeded this time limit for an instance. We measured the time for every solved instance in that time frame, counted the number of timeouts and computed two kinds of average time for all solvers. The first one considers interrupted instances with the cutoff in order to give a feeling how long the solvers actually take including the interrupted instances. For the actual comparison, our interest focuses on the amount of solved instances so that we use a lexicographic order by considering the number of timeouts firstly and the average times without timeouts secondly if the same number of instances have been cancelled. For the case of distinct numbers of timeouts, we only consider these and omit the second average. If two or all solvers share the number of cancelled instances, we fix the leftmost one to be 100 % and calculate the relative average time for all of them. The number of timeouts coincides with the ratio as 100 instances have been employed.

Examination For the sake of readability, we use some abbreviations for this section: the approach using the polynomial algorithm for formulas in CNF is called `CNF`, the one using the lattice of assignment `LoA` while the method incorporating a SAT-Solver is called `SATTS` (SAT-Solver & Tseytin). For the first four configurations, we see that `CNF` is able to keep up with the others considering the timeouts but is outperformed conspicuously considering the solved instances. For almost all

	Conjunctive Normalform				Lattice of Assignment				SAT-Solver			
	with	time-	without		with	time-	without		with	time-	without	
	avg/s	outs	avg/s	%	avg/s	outs	avg/s	%	avg/s	outs	avg/s	%
5/ 5/ 5	3.09	2	1.120	100	2.03	2	0.030	3	2.09	2	0.090	8
5/ 5/10	2.18	2	0.180	100	2.12	2	0.120	68	2.07	2	0.070	41
5/ 5/15	2.50	2	0.510	100	2.12	2	0.120	24	2.35	2	0.360	71
5/ 5/20	3.03	2	1.060	100	2.17	2	0.177	17	3.53	3	-	-
5/10/ 5	6.44	5	-	-	5.49	4	1.560	100	4.69	4	0.720	47
5/10/10	9.15	8	-	-	7.23	7	0.250	100	8.24	7	1.330	535
5/10/15	5.51	5	-	-	4.76	4	0.796	100	5.09	4	1.130	142
10/ 5/ 5	9.62	9	-	-	8.34	8	-	-	6.99	6	-	-
10/ 5/10	5.37	5	-	-	3.70	3	0.730	100	4.38	3	1.420	195
10/ 5/15	4.19	3	1.230	100	3.53	3	0.542	44	2.99	2	-	-
10/10/ 5	6.22	6	-	-	5.34	4	1.400	100	4.74	4	0.770	55
10/10/10	6.37	6	0.389	100	6.13	6	0.138	36	6.37	5	-	-
15/ 5/ 5	1.16	1	0.159	100	1.01	1	0.013	8	1.03	1	0.032	20
15/ 5/10	2.34	2	0.346	100	2.09	2	0.095	28	3.28	3	-	-

Figure 10: The first column corresponds to the parameters: $|Q|/|T|/|N|$. For every solver and configuration, the overall average time is given in the first column. The remaining columns show the number of timeouts and the average time for the solver(s) with the same number of timeouts for a lexicographic order.

other configurations, `CNF` produces more timeouts than at least one of the other two approaches. There might be some outliers like the last configuration where `CNF` has even less timeouts than `SATTS`. However, `LoA` and `SATTS` do not only increase the number of solved instances overall but they are faster than `CNF` for most cases. Therefore, we investigate on the two approaches for non-normalized formulas from now on. We observe the biggest difference for one of the configurations in the middle (`10/ 5/ 5`) for which `SATTS` has solved 94 instances whereas `LoA` has only solved 92. On the opposite side, `LoA` got less timeouts for two other configurations. On the whole, counting the winner for every configuration, i.e. the solver with the best performance according to the lexicographic order, leads to quite a balanced result for `LoA` and `SATTS`. We once observed that there were two different cancelled instances even if the number of these did match. So we assume both approaches to be valuable for different kinds of examples. Real examples for our tool are more relevant than distinguishing and detecting these structures as we might not be able to force these in such a detected structure. This leads to our next chapter in which we present possible directions of future work.

9 Conclusion and Future Work

To conclude the thesis, we give a short recap by pointing at the important ideas and results of the previous chapters. Moreover, we present two starting points for future work and explain their importance.

9.1 Conclusion

After introducing some common notation and concepts in Chapter 2 we proceeded with the definition of positive Boolean formulas for which some basic properties like monotonicity were shown. We also defined the game arena on which the context-free games take place. We extended the definition of formulas to handle relational compositions and explained the summary technique for context-free games. Furthermore, we pointed to the importance of implication checks for the procedure and therefore focused on them for the remainder of the thesis.

In Chapter 3, we proved the problem of entailment checking to be co-NP-complete by giving an algorithm in co-NP and proving the complement problem to be NP-hard with a reduction from 3SAT with non-mixed clauses to non-entailment checking. Moreover, we presented three polynomial algorithms to check entailment for special cases in which both formulas are given in some normal form, i.e. CNF or DNF.

Chapter 4 proposed our first approach to handle compositions in implication checks with a sequent calculus consisting of different inference rules to handle conjunctions, disjunctions and compositions. We proved the sequent calculus to be sound and showed an upper bound for the number of applications of inference rules for conjunction and disjunction to give a proof for a valid sequent and thus proved completeness as well. To conclude the chapter, we commented on different orders of precedence for the application of inference rules.

In Chapter 5, we presented an algorithm to exploit the monotonicity of positive Boolean formulas. We traverse the lattice of assignment for one formula to find minimal or maximal satisfying assignments which are potential counterexamples and check them. We showed that this number can be exponential and thus fits to the co-NP-completeness we have shown before. Moreover, we demonstrated how this basic algorithm can be extended to handle compositions symbolically as well.

Chapter 6 discussed how the process of fixed-point iteration can be exploited. There, the key idea was that a non-terminal is owned by the same player and therefore will stay a conjunction or disjunction at some point. For them, unchanged parts can be omitted on the right side in case of conjunctions and on the left side in case of disjunctions. Our second attempt to split compositions and handle them separately has not been successful and thus not been applicable.

We illustrated the structure of our tool in Chapter 7 that applies the fixed-point iteration presented in Chapter 2. After we explained the parameters for the random

generator, we described the preprocessor which also uses a fixed-point iteration to compute the set of non-terminals remaining *false* all the time. We proceeded with the implementation of the actual summary technique consisting of the solver, an iterator and the formula manager and its box manager. We also exhibited how we implemented the theory of checking implication for formulas without compositions and pointed to challenges of implementing the theory for formulas with compositions.

Consequently, we presented our benchmarking results in Chapter 8 that demonstrated the performance of implication checks for non-normalized formulas. The approach using the lattice of assignment can compete with the procedure incorporating a SAT-Solver and each of them seems to outperform the other one for different kinds of examples. Both improved the overall results compared to the algorithm for CNF as we spotted a number of solved instances which have been infeasible with an algorithm for normalized formulas, i.e. the one for CNF.

9.2 Future Work

Exploiting Kleene Iteration Our motiviation for implication checks is their application in the summary technique from [HMM16]. In order to enforce termination of the fixed-point iteration, we need to check whether two formulas are logically equivalent. The Kleene iteration guarantees that the old formula implies the new one. Based on this assumption, checking whether the new formula implies the old one is a special case. So far, we did not find any means to profit from this fact, but it might be a good starting point to find optimizations and heuristics.

Real Examples The examples in the presented tool are currently generated randomly. In order to get closer to the overall goal, synthesis of software and hardware, it is essential to have translations of such problems to our domain. We have sketched how the elements of a basic programming language could be encoded into context-free games. So in general, it should be possible to develop a small kernel of a programming language which fits into the current framework to generate actual software synthesis problems. On the other hand, considering hardware synthesis there are some specification languages, but the translation is non-trivial. Despite of the bridge between our tool and a practical example, real examples are crucial for the further development of this tool as well. Usually heuristics only increase the speed of cases with a specific structure while cases different from them have a worse performance than before. E.g. optimizing the case of non-implication will lead to slower performance in the case of implication as we explained in the sequent calculus for example. In general, the key to success is to improve the average performance. Obviously, most sophisticated heuristics could be counterproductive if they improve cases which do not occur in practice.

Reducing Formulas During the experiments we realised that some formulas are kind of redundant. Because of the Kleene iteration, this redundancy will carry over in every step and thus the formulas will increase excessively. Therefore, it

could be advantageous to use techniques similar to subsumption in SAT-Solvers to reduce the formulas. One possibility is the usage of the simulation relation from [ACC+11] to minimize the formulas after each or some steps of the fixed-point iteration. Additionally, one could use the same simulation to weaken the notion of implications.

List of Figures

References

[ACC+11] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. Advanced Ramsey-Based Büchi Automata Inclusion Testing. In *CONCUR*, volume 6901 of *LNCS*, pages 187–202, 2011.

[AZ98] Martin Aigner and Günter M. Ziegler. *Proofs from the Book*. Springer, 1998.

[BBCB16] Jaroslav Bendík, Nikola Benes, Ivana Cerná, and Jirí Barnat. Tunable Online MUS/MSS Enumeration. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, pages 50:1–50:13, 2016.

[Bus98] Samuel Buss. *An Introduction to Proof Theory*. Elsevier, 1998.

[DG93] James P. Delgrande and Arvind Gupta. Two Results in Negation-free Logic. In *Applied Mathematics Letters*, pages 79 – 83, 1993.

[DG96] James P. Delgrande and Arvind Gupta. The Complexity of Minimum Partial Truth Assignments and Implication in Negation-free Formulae. In *Annals of Mathematics and Artificial Intelligence 18*, pages 51 – 67, 1996.

[Grä98] George Grätze. *General Lattice Theory*. Springer, 1998.

[HMM16] Lukás Holík, Roland Meyer, and Sebastian Muskalla. Summaries for Context-Free Games. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, pages 41:1–41:16, 2016.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Kre98] Christoph Kreitz. Program Synthesis. In *Automated Deduction – A Basis for Applications*, pages 105–134, 1998.

[MRW05] Peter Bro Miltersen, Jaikumar Radhakrishnan, and Ingo Wegener. On converting CNF to DNF. In *Theoretical Computer Science*, pages 325 – 335, 2005.

[Sch78] Thomas J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the 10th annual ACM symposium on Theory of computing*, pages 216 – 226, 1978.

[Sip05] Michael Sipser. *Introduction to the Theory of Computation*. Wadsworth Inc Fulfillment, 2nd edition, 2005.

[Tse70] Grigorii S. Tseytin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. A.O. Slisenko, 1970.