Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

# Correctness of Tableau-Based Decision Procedures with Backjumping

submitted by

**Tobias Tebbi**

submitted on

**5.4.2011**

Supervisor

**Prof. Dr. Gert Smolka**

Advisors

**Mark Kaminski, M.Sc.**

**Prof. Dr. Gert Smolka**

Reviewers

**Prof. Dr. Gert Smolka**

**Prof. Bernd Finkbeiner, Ph.D.**

## Eidesstattliche Erklärung

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

## Statement in Lieu of an Oath

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

## Einverständniserklärung

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

## Declaration of Consent

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

Saarbrücken, ____________________ ____________________

Date Signature

# Abstract

Implementations of tableau-based decision procedures often use an optimization called backjumping. The goal of this thesis is to prove the correctness of the backjumping optimization. To do so, we define an abstract class of terminating tableau systems and show the correctness of a concomitant decision procedure performing depth-first search with backjumping. Based on this framework we obtain the correctness of a backjumping decision procedure for modal logic. To the best of our knowledge, this is the first rigorous correctness proof for such a procedure.

# Acknowledgment

I would like to express my thankfulness for my advisors Mark Kaminski and Gert Smolka. The countless discussions, advices and suggestions have helped me profoundly to develop the results and to write the thesis. This allowed me to gain invaluable insight in the subject as well as in scientific writing in general.

I would also like to thank Gert Smolka and Bernd Finkbeiner for reviewing this thesis.

# Contents

**Introduction** **11**

**1 The Propositional Case** **15**
1.1 Tableau Systems . . . 15
1.1.1 Fundamental Notions . . . 15
1.1.2 Termination . . . 16
1.1.3 Correctness . . . 16
1.1.4 Comparison to Other Propositional Tableau Systems . . . 18
1.1.5 Compact Notation For Tableaux . . . 18
1.2 Depth-First Search . . . 19
1.3 Backjumping . . . 20

**2 Abstract Tableau systems** **23**
2.1 Rules . . . 23
2.2 Consistency . . . 25
2.3 Termination . . . 26

**3 Correctness of Depth-First Search** **27**

**4 Correctness of Backjumping** **31**
4.1 Dependency Sequents . . . 31
4.1.1 Example . . . 31
4.1.2 Relation to Propositional Logic . . . 34
4.1.3 Dependency Sequent Rules . . . 34
4.2 DFS with Backjumping . . . 35
4.3 Learning . . . 40

**5 The Modal Case** **41**
5.1 Syntax and Semantics of K . . . 41
5.2 Rules . . . 42
5.3 Example . . . 44

**6 Discussion and Future Work** **49**
6.1 Blocking . . . 49
6.2 Non-terminating Tableau Systems . . . 49
6.3 Monotonicity . . . 49

# Introduction

Tableau methods are used for various purposes in logic, especially as the basis for decision procedures. Fitting [6] sees tableau methods as formal proof procedures that try to refute a formula by "[breaking it] down syntactically, generally splitting things into several cases". They were invented by Beth [4] and Hintikka [12] and later refined by Lis [22] and Smullyan [29]. Originally developed as proof procedures for first-order logic, they were later extended to modal logics [7], description logics [1], higher-order logic [2] and many more (see [6]).

Tableau methods have found use as decision procedures in theorem provers for modal and description logics (e.g., FaCT++ [31], HTab [13], RACER [11] or Spartacus [10]). These implementations use optimization techniques that are essential for the practical performance. A particularly important optimization technique, used by all of the above implementations, is backjumping.

Backjumping is a search optimization technique originally developed for constraint satisfaction problems (CSPs). We will now give an example of backjumping for a CSP, the satisfiability problem for propositional logic. Assume, for instance, we search a satisfying assignment of the propositional formula $(x_1 \vee x_4) \wedge (x_1 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$. Every clause is a constraint on a satisfying assignment. In our case, every assignment where $x_1 = 1$ is a solution.

The search space of a search problem can be represented by a search tree. In our case, the search tree is a decision tree. Every node corresponds to a partial assignment. For a node that is labeled with $x_i$, the nodes in the left subtree correspond to assignments that set $x_i$ to 0 and the nodes in the right subtree correspond to assignments that set $x_i$ to 1. For example, Figure 0.1 is a search tree for $(x_1 \vee x_4) \wedge (x_1 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$.

If a node violates a constraint, then it is labeled with $\perp$ and called **failed**. We search for a node satisfying all constraints, which is labeled with $\top$. The usual approach is to perform a depth-first search in the search tree. This is indicated with dotted arrows in Figure 0.1. When the search reaches a failed node, then it walks up in the tree until it comes to a node with an unvisited child. This process is called backtracking. The whole algorithm, that is, depth-first search in the search tree of a CSP, is also called backtracking search [27].

Backjumping is an improvement of backtracking search. It prevents exploring certain useless parts of the search tree. For example, consider the subtree in Figure 0.1 where $x_1 = 0$. It cannot contain a solution because every assignment

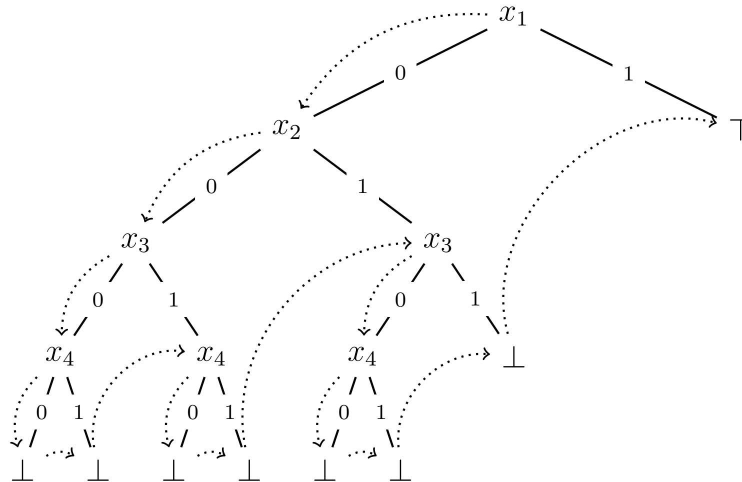


Figure 0.1: Search tree for the formula $(x_1 \vee x_4) \wedge (x_1 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$

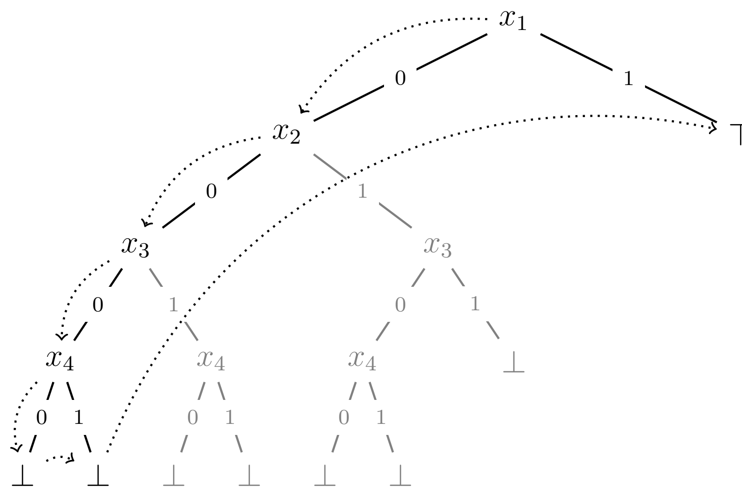


Figure 0.2: Search tree from Figure 0.1 with backjumping

where $x_1 = 0$ violates either $x_1 \vee x_4$ or $x_1 \vee \neg x_4$, depending on the value of $x_4$. For this reason, trying different values for $x_2$ and $x_3$, as done by naive backtracking search in Figure 0.1, is a waste of time since it cannot possibly yield a satisfying assignment.

Backjumping improves backtracking search by analyzing the reasons for failures. In the case at hand, it determines that $x_2$ and $x_3$ are independent from the failures in the first two leaves. Consequently, it does not try other values for $x_2$ and $x_3$ but directly "jumps back" in the tree to the node where $x_1$ is set to a different value, that is, 1. This is depicted in Figure 0.2.

The basic ideas of backjumping were introduced in different forms by Gaschnig [8] and Stallman and Sussman [30]. The version of backjumping we are going to consider is the one presented by Prosser [26] and Ginsberg [9]. Ginsberg also gives a formal proof of the correctness of backjumping for CSPs. As the propositional satisfiability problem can be seen as a CSP, backjumping can be used in DPLL-style theorem provers. Nieuwenhuis et al. [25] give a correctness proof of backjumping for a DPLL-style calculus that does not rely on a translation from propositional satisfiability to a CSP. Backjumping has also been adapted to tableau methods for modal logics and description logics [14, 15, 17, 19]. It has been observed to yield a considerable performance improvement for reasoners deciding these logics [16, 19]. Although backjumping has been used in the implementation of many tableau-based theorem provers (see above), its correctness for tableau procedures was only argued informally relying on the intuition that modal satisfiability can be represented as a CSP.

## Contributions

The goal of this thesis is to give a formal correctness proof of backjumping in the context of tableau systems. For this, we define an abstract framework of tableau systems and use it to give a formal description of a tableau procedure with backjumping. Then we prove this procedure to be correct. This way, one obtains a correct tableau procedure with backjumping for every tableau system that is an instance of the abstract framework. As an example of such an instance, we present a tableau system for the basic modal logic $K$. To the best of our knowledge, this is the first rigorous correctness proof for a tableau procedure with backjumping for modal logic.

## Overview

In Chapter 1, we informally introduce the fundamental concepts of this thesis. We present a tableau system for propositional logic and use it to explain how to build a tableau using a depth-first strategy and backjumping.

In Chapter 2, we define an abstract framework of tableau systems and discuss how the propositional tableau system from Chapter 1 fits into this framework.

In Chapter 3, we present depth-first search for tableaux as a recursive procedure and prove its correctness.

In Chapter 4, we first introduce the notion of a dependency sequent, which expresses dependencies between nondeterministic decisions and inconsistencies or formulas on a tableau. Then we use this notion to formulate a tableau algorithm with backjumping as a recursive procedure. Finally, we prove this procedure to be correct.

In Chapter 5, we present a tableau system for the basic modal logic $K$ as a second example of a tableau system that fits into our abstract framework besides the propositional tableau system from Chapter 1.

# 1 The Propositional Case

In this chapter, we will use an example tableau system for propositional logic to give an overview of the techniques and algorithms used in this thesis.

## 1.1 Tableau Systems

### 1.1.1 Fundamental Notions

A **tableau** is a binary tree whose nodes are labeled with formulas. For a node with two child nodes, the first child node is called the **left child** and the second one the **right child**. The paths from the root to a leaf are called the **branches** of the tableau. We say **a formula is on a branch** if the branch contains a node labeled with the formula. Tableaux are constructed using **tableau rules**. We consider rules of the form $\frac{s_1,\cdots,s_n}{}$, $\frac{s_1,\cdots,s_n}{t}$ or $\frac{s_1,\cdots,s_n}{t_1 \mid t_2}$ where $s_1, \ldots, s_n, t, t_1$ and $t_2$ are formulas. The set of formulas above the line is called **premise**, the formulas below build the set of up to two possible **conclusions**. The rules are used to extend a branch of a tableau. When a rule is applied to a branch, then the conclusions of the rule are appended as new children to the leaf of the branch. A rule only applies to a branch if the branch contains all formulas of the premise and none of the conclusions. Note that this ensures that a branch contains a formula at most once. A rule with two conclusions is called **branching** because it transforms one branch into two branches. A rule with zero conclusions is called **clashing**. When a clashing rule applies to a branch, then the branch is called **clashed**. A branch to which no rule applies is called **evident.** A **tableau for a formula** $s$ is a tableau built using the rules of the tableau system starting with $s$, i.e., the tableau with a single node labeled with $s$. A **tableau system** consists of a set of tableau rules.

Figure 1.1 shows a tableau system for negation-normal propositional formulas. The rules are to be read as schemas. Every rule schema stands for infinitely many rule instances, which are obtained from the schema by replacing the metavariables

$$T_\wedge \frac{s_1 \wedge \cdots \wedge s_n}{s_i}\ {\scriptstyle i\in[1,n]} \qquad T_\vee \frac{s_1 \vee s_2}{s_1 \mid s_2} \qquad T_\otimes \frac{s, \neg s}{}$$

Figure 1.1: Rules of a tableau system for propositional logic

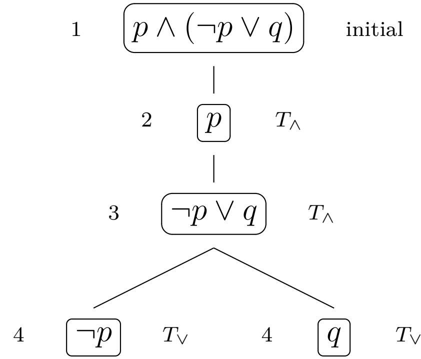


Figure 1.2: Tableau for $p \wedge (\neg p \vee q)$

$s_1, \ldots, s_n$ with propositional formulas. For example, $\frac{p \wedge q}{p}$ and $\frac{p \wedge q \wedge \neg q}{q}$ are instances of the schema $T_\wedge$. Instances of $T_\vee$ are branching, instances of $T_\otimes$ are clashing. An example of a tableau for the formula $p \wedge (\neg p \vee q)$ is shown in Figure 1.2. It consists of two branches. The left branch is clashed because the rule $T_\otimes$ applies to it. The right branch is evident. The numbers on the left denote the order in which the tree nodes are added. On the right, the tableau rule used to add the respective formula is given. Note that $T_\vee$ always adds two formulas at the same time.

### 1.1.2 Termination

The purpose of a tableau system is to serve as a proof procedure for a logic. We will only consider terminating tableau systems in this thesis. A tableau system is terminating if it is impossible to extend a tableau infinitely often. This means that after finitely many steps, all branches are clashed or evident. We call such a tableau **maximal**.

The propositional tableau system terminates because every branch only contains subformulas of the initial formula. Thus the length of a branch is bounded by the number of subformulas of the initial formula because a branch contains a formula at most once. Binary trees with bounded depth have bounded size.

### 1.1.3 Correctness

Terminating tableau systems are usually used to decide the satisfiability of formulas. This is done as follows. One builds a tableau for a formula $s$ by stepwise applying the rules of the tableau system. This can be seen as a search for an evident branch. If the search succeeds, that is, one has built a tableau containing an evident branch, then one concludes $s$ to be satisfiable. Else, one can extend the

tableau until it consists only of clashed branches and conclude $s$ to be unsatisfiable. Such a tableau is called **clashed.** This reasoning can be shown to be correct for the tableau system in Figure 1.1 by proving the following two properties.

**refutational soundness**: A formula that has a clashed tableau is unsatisfiable.
So a clashed tableau for a formula $s$ is a proof that $s$ is unsatisfiable. As an example, consider the clashed tableau for the formula $p \wedge (\neg p \vee q) \wedge (\neg q \vee \neg p)$ depicted in Figure 1.3. The tableau consists of three branches. All of them are clashed because the rule $T_\otimes$ applies to them. Thus, the tableau is a proof for the unsatisfiability of $p \wedge (\neg p \vee q) \wedge (\neg q \vee \neg p)$.
We sketch the proof of the contraposition of refutational soundness: A satisfiable formula has no clashed tableau.
We require that if a tableau rule applies to a satisfiable branch, then at least one of its conclusions yields a satisfiable extension of the branch. One easily verifies this for the rules in Figure 1.1. For example, look at the rule $T_\vee$. If a satisfiable branch $B$ contains $s \vee t$, then either $B$ extended with $s$ or $B$ extended with $t$ is satisfiable.
Since clashing rules have no conclusions, they must not apply to satisfiable branches.
Following [29], we give semantics to tableaux. We see a branch as the conjunction of the formulas it contains and a tableau as the disjunction of its branches. Therefore we call a branch satisfiable if the conjunction of all formulas on this branch is satisfiable and we call a tableau satisfiable if it contains a satisfiable branch.
Because the rules maintain the satisfiability of branches, they also maintain the satisfiability of tableaux. Thus if $s$ is satisfiable, then every tableau for $s$ is satisfiable. A satisfiable tableau contains a satisfiable branch, which is not clashed because a clashing rules cannot apply to a satisfiable branch. So a satisfiable formula has no clashed tableau.

**refutational completeness**: If $s$ is unsatisfiable, then all maximal tableaux for $s$ are clashed.
Again, we sketch the proof of the contraposition: If s has a maximal tableau that contains an evident branch, then $s$ is satisfiable. This is the contraposition because every maximal tableau that is not clashed contains an evident branch.
For the propositional tableau system, one can show that evident branches are satisfiable by constructing a propositional model from the evident branch. Since every branch of the tableau contains the initial formula, the existence of a satisfiable branch implies that the initial formula is satisfiable.
As an example, consider the tableau in Figure 1.2 presented before.

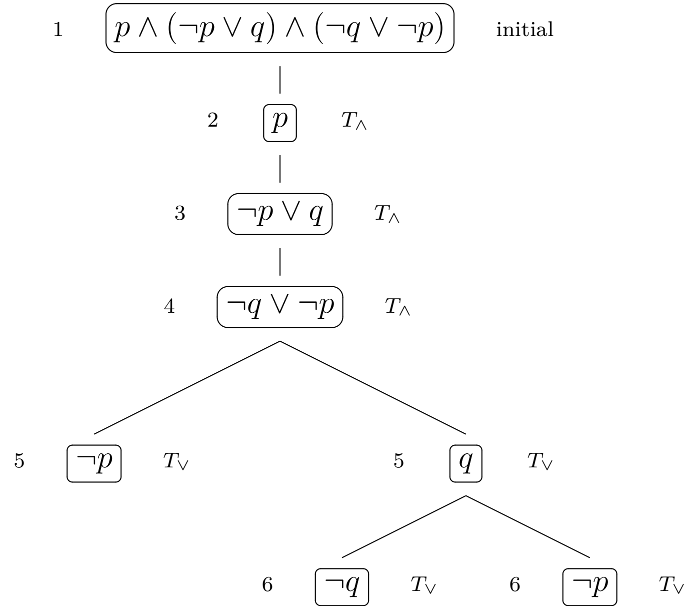


Figure 1.3: Tableau for $(p_1 \vee p_2) \wedge (q \wedge \neg q)$

Its right branch is evident. Thus, the tableau proves that the formula $p \wedge (\neg p \vee q)$ is satisfiable.

### 1.1.4 Comparison to Other Propositional Tableau Systems

We use a rule for n-ary conjunctions to obtain smaller examples. Usually (e.g., Smullyan [29]), one uses a rule for binary conjunctions and writes it as $\frac{s_1 \wedge s_2}{s_1,\, s_2}$. Note the additional difference that this rule adds two formulas to the branch at once. To simplify the formalization, we want a tableau rule to add exactly one formula to every new branch at a time. These differences are inessential because an application of $\frac{s \wedge t}{s,\, t}$ can be simulated by applying the rule $T_\wedge$ twice.

### 1.1.5 Compact Notation For Tableaux

So far, we have drawn tableaux as one usually draws trees. However, for large examples this quickly becomes cumbersome. Therefore we will use a more compact notation for tableaux. Figure 1.4 represents the same tableau as Figure 1.3. The symbol $\otimes$ is used to mark clashed branches.

$$p \wedge (\neg p \vee q) \wedge (\neg q \vee \neg p)$$
$$p$$
$$\neg p \vee q$$
$$\neg q \vee \neg p$$

| $\neg p$ | $q$ | |
|---|---|---|
| $\otimes$ | $\neg q$ | $\neg p$ |
| | $\otimes$ | $\otimes$ |

Figure 1.4: A more compact notation for tableaux

| | |
|---|---|
| 1 | $(q_1 \vee \neg q_2) \wedge q_2 \wedge (p_1 \vee p_2) \wedge (\neg q_1 \vee \neg q_2)$ |
| 2 | $q_1 \vee \neg q_2$ |
| 3 | $q_2$ |
| 4 | $p_1 \vee p_2$ |
| 5 | $\neg q_1 \vee \neg q_2$ |

| 6 $q_1$ | | | | 6 $\neg q_2$ |
|---|---|---|---|---|
| 7 $p_1$ | | 7 $p_2$ | | 14 $\otimes$ |
| 8 $\neg q_1$ | 8 $\neg q_2$ | 11 $\neg q_1$ | 11 $\neg q_2$ | |
| 9 $\otimes$ | 10 $\otimes$ | 12 $\otimes$ | 13 $\otimes$ | |
| $a$ | $b$ | $c$ | $d$ | $e$ |

Figure 1.5: A clashed tableau built with DFS

## 1.2 Depth-First Search

How can we build tableaux systematically? One usually uses a particular strategy to extend tableaux: depth-first search (DFS). DFS is a well-known search strategy in graphs. For tableaux, it can be realized as follows. One always extends the leftmost branch that is not clashed. When this branch cannot be extended because it is evident, then the procedure stops because the evident branch proves that the initial formula is satisfiable. If all branch are clashed, then the tableau is clashed and hence the procedure stops because the clashed tableau proves the initial formula to be unsatisfiable.

DFS is used because it is memory efficient. As the contents of clashed branches do not influence the result of the procedure, one only has to store the open branches. For tableaux produced by DFS, the number of nodes in open branches is linear in the depth of the tableau.

As an example of DFS, look at the clashed tableau in Figure 1.5. The numbers on the left denote the order in which formulas are added and clashes are detected. The tableau has five clashed branches $(a, b, c, d, e)$. DFS first explores the leftmost branch $a$ until it detects a clash in step 9. Then it tries to explore the next branch

$b$ but directly detects a new clash in step 10. Then, it explores the branches $c$, $d$ and $e$ in the same way.

Note that branch $a$ is similar to branch $c$ and branch $b$ is similar to branch $d$. The only difference is that $a$ and $b$ contain $p_1$ while $c$ and $d$ contain $p_2$. But $p_1$ and $p_2$ do not contribute to any clash. The clashes in branch $a$ and branch $b$ are independent of $p_1$ and so the same clashes reappear in branch $c$ and branch $d$. Thus, exploring $c$ and $d$ is unnecessary work, which backjumping will allow us to avoid.

## 1.3 Backjumping

Backjumping is an optimization technique which aims at pruning the search space of a search procedure. In the context of tableaux, backjumping is an optimization of DFS that allows that certain unsatisfiable branches do not need to be explored. Backjumping avoids the exploration of branches that differ only in inessential details from branches that were already determined to be clashed. For DFS, when we detect a clash, then we continue exploration at the next possible branching point above the clash. If the clash is independent of the formula added to the clashed branch at this branching point, then we know that also the unexplored branch of this branching point is unsatisfiable. Therefore we do not need to explore this branching point any further and can continue with the next branching point above. This is what backjumping does. In other words, when backtracking from a clash, we jump over branching points that are independent of the clash.

In order to find irrelevant branching points, we need to determine the branching points a clash depends on. Since clashes are derived from formulas, we trace the branching points a formula depends on for every formula on the tableau. This can be done as follows. We associate an index with every branching point. For every formula, we store the set of indices of branching points the formula depends on. These sets are called **dependency sets**. Backjumping works by performing a DFS and simultaneously deriving dependency sets for all new formulas and clashes. The dependency set of a clash is used to determine irrelevant branching points, which are then not explored further.

For an example of backjumping, consider the tableau in Figure 1.6, which avoids the unnecessary work done in the tableau in Figure 1.5. The three branching points have the indices 1, 2 and 3. On the left of every formula, the respective dependency set is given.

The initial formula is always assigned the dependency set $\{0\}$. A formula derived from a conjunction inherits the dependency set of the conjunction. Therefore $q_1 \vee \neg q_2$, $q_2$, $p_1 \vee p_2$ and $\neg q_1 \vee \neg q_2$ also obtain the dependency set $\{0\}$. The left formula of a branching point with index $i$ is always assigned the dependency set $\{i\}$. This applies to $q_1$, $p_1$ and $\neg q_1$. The dependency set of a clash is the union of

$$\begin{array}{lll}
0 & \Rightarrow & (q_1 \vee \neg q_2) \wedge q_2 \wedge (p_1 \vee p_2) \wedge (\neg q_2 \vee \neg q_1) \\
0 & \Rightarrow & q_1 \vee \neg q_2 \\
0 & \Rightarrow & q_2 \\
0 & \Rightarrow & p_1 \vee p_2 \\
0 & \Rightarrow & \neg q_1 \vee \neg q_2
\end{array}$$

Branching point 1: left $1 \Rightarrow q_1$ | right $0 \Rightarrow \neg q_2$, $0 \Rightarrow \otimes$ (branch $d$)

Branching point 2 (below $1 \Rightarrow q_1$): left $2 \Rightarrow p_1$ | right $p_2$ (branch $c$)

Branching point 3 (below $2 \Rightarrow p_1$): left $3 \Rightarrow \neg q_1$, $1,3 \Rightarrow \otimes$ (branch $a$) | right $0,1 \Rightarrow \neg q_2$, $0,1 \Rightarrow \otimes$ (branch $b$)

$a \qquad b \qquad c \qquad d$

Figure 1.6: A tableau refutation with backjumping

the dependency sets of the two clashing terms. Thus the clash in branch $a$ is the union of the dependency sets of $q_1$ and $\neg q_1$, that is, $\{1, 3\} = \{1\} \cup \{3\}$.

The right-hand formula of a branching point $i$ can first obtain a dependency set when the subtree of the left-hand formula is finished. It is labeled with the union of the dependency set of the disjunction which caused the branching point and the dependency set of the last clash in the left subtree except for $i$. So the dependency set of $\neg q_2$ in branch $b$ is $\{0, 1\} = \{0\} \cup (\{1, 3\} - \{3\})$, which is the union of the dependency set of the disjunction $\neg q_1 \vee \neg q_2$ and the dependency set of the clash in branch $a$ except for 3. The disjunction implies that either the left-hand or the right-hand formula is a satisfiable extension of branch above the branching point. The clash showed that the left alternative of the branching point yields an unsatisfiable branch. Thus the right-hand formula must be a satisfiable extension. This is why the right-hand formula depends on the clash.

The clash in branch $b$ is assigned the dependency set $\{0, 1\} = \{0\} \cup \{0, 1\}$ because it is derived from the formulas $q_2$ and $\neg q_2$. Now something interesting happens. DFS without backjumping would continue exploration with branch $c$, which is the right-hand side of branching point 2. But because the dependency set of the clash does not contain 2, one can conclude that the clash is independent of branching point 2. Thus it is unnecessary to explore branch $c$. For this reason, DFS with backjumping directly continues with branch $d$.

Backjumping can result in an exponential speed-up of the search. An example for this is the following formula.

$$(p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge \cdots \wedge (p_{2n-1} \vee p_{2n}) \wedge (\neg q_1 \vee \neg q_2) \wedge q_1 \wedge q_2$$

If DFS first expands the $n$ irrelevant disjunctions $p_i \vee p_{i+1}$ before analyzing the unsatisfiable part $(\neg q_1 \vee \neg q_2) \wedge q_1 \wedge q_2$, then one obtains a huge tableau. It is a

$$
\begin{array}{lll}
0 & \Rightarrow & (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge \cdots \wedge (p_{2n-1} \vee p_{2n}) \wedge (\neg q_1 \vee \neg q_2) \wedge q_1 \wedge q_2 \\
0 & \Rightarrow & p_1 \vee p_2 \\
0 & \Rightarrow & p_3 \vee p_4 \\
\vdots & \vdots & \vdots \\
0 & \Rightarrow & p_{2n-1} \vee p_{2n} \\
0 & \Rightarrow & \neg q_1 \vee \neg q_2 \\
0 & \Rightarrow & q_1 \\
0 & \Rightarrow & q_2
\end{array}
$$

1: $1 \Rightarrow p_1$ | $p_2$

2: $2 \Rightarrow p_3$ | $p_4$

3: $\vdots$ | $\ddots$ | $p_6$

$n-1 \Rightarrow p_{2n-3}$

$n$: $n \Rightarrow p_{2n-1}$ | $p_{2n}$

$n+1$: $n+1 \Rightarrow \neg q_1$, $0, n+1 \Rightarrow \otimes$ | $0 \Rightarrow \neg q_2$, $0 \Rightarrow \otimes$

Figure 1.7: A tableau refutation where backjumping saves exponential work

balanced binary tree of depth $\geq n$ and size $\geq 2^n$. Although tableaux of exponential size are inevitable in general due to the computational hardness of propositional logic, backjumping always solves this particular formula in linear time. Consider the tableau in Figure 1.7. In the second branch, we obtain a clash that has just the dependency set $\{0\}$, which means that all remaining branching points are irrelevant and do not need to be explored.

# 2 Abstract Tableau systems

So far, we introduced tableau systems only informally. The informal notions from before will now be defined formally.

We assume some set $\mathcal{F}$ of formulas to be fixed. (For propositional logic, $\mathcal{F}$ is the set of all propositional formulas.)

The formalization of tableaux will not model trees explicitly. Instead we will only formalize branches and state all properties regarding tableaux as properties of branches. We define a **branch** as a finite nonempty set of formulas $B \subseteq \mathcal{F}$. Note that unlike our informal notion of branches, the formal definition represents the formulas on a tableau branch without any additional structure.

## 2.1 Rules

We model **tableau rules** as pairs $\frac{P}{C}$ of a premise $P$ and a set of possible conclusions $C$.

**Definition 2.1.** A **tableau system** is a set of tableau rules $\mathcal{T}$ with

$$\mathcal{T} \subseteq \{\tfrac{P}{C} \mid P, C \subseteq \mathcal{F} \text{ where } P \text{ is finite and } |C| \leq 2\}$$

We fix some tableau system $\mathcal{T}$. All following definitions are relative to $\mathcal{T}$.

The informal presentation of rule schemas like the ones depicted in Figure 1.1 corresponds to formal tableau rules as shown in the following table. Recall that a rule schema corresponds to infinitely many rule instances.

$$\frac{s_1 \wedge \cdots \wedge s_n}{s_i}\; i\in[1,n] \quad \rightsquigarrow \quad \{\tfrac{\{s_1 \wedge \cdots \wedge s_n\}}{\{s_i\}} \mid s_1, \ldots, s_n \in \mathcal{F}, i \in [1, n]\} \subseteq \mathcal{T}$$

$$\frac{s_1 \vee s_2}{s_1 \mid s_2} \quad \rightsquigarrow \quad \{\tfrac{\{s_1 \vee s_2\}}{\{s_1, s_2\}} \mid s_1, s_2 \in \mathcal{F}\} \subseteq \mathcal{T}$$

$$\frac{s, \neg s}{\phantom{x}} \quad \rightsquigarrow \quad \{\tfrac{\{s, \neg s\}}{\{\}} \mid s \in \mathcal{F}\} \subseteq \mathcal{T}$$

To define when a tableau rule applies to a branch, we introduce the notion of steps. A step $\frac{B}{C}$ models the fact that the tableau system allows to extend the branch $B$ with the conclusions $C$.

**Definition 2.2.** We define $\tilde{\mathcal{T}} := \{\frac{B}{C} \mid B \text{ is a branch}, \frac{P}{C} \in \mathcal{T} \text{ for some } P \subseteq B\}$.

The **step closure** $\hat{\mathcal{T}}$ of a tableau system $\mathcal{T}$ is defined as

$$\hat{\mathcal{T}} := \{\tfrac{B}{C} \in \tilde{\mathcal{T}} \mid C = \{\} \text{ or } \tfrac{B}{\{\}} \notin \tilde{\mathcal{T}} \text{ and } C \cap B = \{\}\}$$

Elements of $\hat{\mathcal{T}}$ are called **steps.**

A rule $\frac{P}{C} \in \mathcal{T}$ **applies to a branch** $B$ if there is a step $\frac{B}{C} \in \hat{\mathcal{T}}$ with $P \subseteq B$.

A branch $B$ is called **evident** if there is no step $\frac{B}{C} \in \hat{\mathcal{T}}$ and **clashed** if $\frac{B}{\{\}} \in \hat{\mathcal{T}}$.

In other words, a clashing rule $\frac{P}{\{\}}$ applies to a branch $B$ if $P \subseteq B$ and a non-clashing rule $\frac{P}{C}$ applies to a branch $B$ if $P \subseteq B$, $B$ is not clashed and does not contain any of the possible conclusions in $C$.

For instance, the tableau rule $\frac{\{s_1 \vee s_2\}}{\{s_1, s_2\}}$ contributes the set of steps

$$\{\frac{B}{\{s_1, s_2\}} | B \text{ is a branch}, s_1 \vee s_2 \in B, \tfrac{B}{\{\}} \notin \tilde{\mathcal{T}}, B \cap \{s_1, s_2\} = \{\}\}$$

Note that instances of branching rule schemas like $\frac{s_1 \vee s_2}{s_1 \mid s_2}$ can be non-branching if the conclusions are identical. For example, we have the formal tableau rule $\frac{\{p \vee p\}}{\{p\}}$. This cannot cause harm because if such a rule was applied as a branching rule, it would lead to two identical branches.

We require the conclusions of a step to be disjoint from the premise to make sure every step enlarges the current branch for every possible conclusion. Else we could apply rules without making any progress. For instance, if $\frac{\{s \vee t, s\}}{\{s, t\}}$ was a step, then we would be able to build the following infinite tableau.

$$\begin{array}{c}
s \vee t \\
\hline
\begin{array}{c|c}
\begin{array}{c} s \\ \hline \begin{array}{c|c} \begin{array}{c} s \\ \hline \begin{array}{c|c} \begin{array}{c} s \\ \hline \begin{array}{c|c} \iddots & t \end{array} \end{array} & t \end{array} \end{array} & t \end{array} \end{array} & t
\end{array}
\end{array}$$

**Proposition 2.3** (Monotonicity).

1. *Let* $\frac{B}{\{\}} \in \tilde{\mathcal{T}}$ *and* $B \subseteq B'$ *for some branch* $B'$. *Then* $\frac{B'}{\{\}} \in \hat{\mathcal{T}}$.
2. *Let* $\frac{B}{C} \in \tilde{\mathcal{T}}$ *and* $B \subseteq B'$ *as well as* $C \cap B' = \{\}$ *for some branch* $B'$ *that is not clashed. Then* $\frac{B'}{C} \in \hat{\mathcal{T}}$.

*Proof.*

1. Let $B$ and $B'$ be as requested. By the definition of $\tilde{\mathcal{T}}$, there is some $P \subseteq B$ such that $\frac{P}{\{\}} \in \mathcal{T}$. Because $P \subseteq B'$, it follows that $\frac{B'}{\{\}} \in \tilde{\mathcal{T}}$. By the definition of $\hat{\mathcal{T}}$, we have $\frac{B'}{\{\}} \in \hat{\mathcal{T}}$.

2. Let $B$ and $B'$ be as requested. By the definition of $\tilde{\mathcal{T}}$, there is some $P \subseteq B$ such that $\frac{P}{C} \in \mathcal{T}$. Because $P \subseteq B'$, it follows that $\frac{B'}{C} \in \tilde{\mathcal{T}}$. Since $C \cap B' = \{\}$ and $B'$ is not clashed, we have $\frac{B'}{C} \in \hat{\mathcal{T}}$. □

**Lemma 2.4.** *If a branch $B$ is evident, then for all steps $\frac{B'}{C} \in \tilde{\mathcal{T}}$ with $B' \subseteq B$, $C \cap B \neq \{\}$.*

*Proof.* Let $B$ be an evident branch. Because $B$ is evident, it is not clashed. Thus for every $\frac{B'}{C} \in \hat{\mathcal{T}}$ where $B' \subseteq B$, we have that $C \cap B \neq \{\}$ because else $\frac{B}{C} \in \hat{\mathcal{T}}$ by Proposition 2.3 contradicting the evidence of $B$. □

## 2.2 Consistency

**Definition 2.5.** A branch $B$ is called **consistent** if there is an evident branch $B' \supseteq B$ and **inconsistent** otherwise. A formula $s$ is called **consistent** if the branch $\{s\}$ is consistent.

**Proposition 2.6.** *Every evident branch is consistent. If a branch $B$ is consistent, then every subset $B' \subseteq B$ is consistent.*

A branch is consistent iff it can be extended to an evident branch. Note that the definition of a consistent formula matches our previous informal notion of consistency.

What about inconsistency? We would like to show that every formula that has a clashed tableau is inconsistent. Since we do not want to formalize tableaux, we instead show the following statements about branches.

**Lemma 2.7.** *Every clashed branch is inconsistent.*

*Proof.* Let $B$ be a clashed branch. Then we have $\frac{B}{\{\}} \in \hat{\mathcal{T}}$. Suppose, for contradiction, that there is an evident branch $B' \supseteq B$. By Lemma 2.4, we have $B' \cap \{\} \neq \{\}$. Contradiction. □

**Lemma 2.8.** *If there is a step $\frac{B}{C} \in \hat{\mathcal{T}}$ such that for all $t \in C$, the branch $B \cup \{t\}$ is inconsistent, then $B$ is inconsistent.*

*Proof.* Let $\frac{B}{C} \in \hat{\mathcal{T}}$ be as required. We show that no branch $B' \supseteq B$ is evident. Let $B' \supseteq B$ be a branch. We make a case distinction.

- If $B' \cap C = \{\}$, then $B'$ is not evident by Lemma 2.4.
- If $B' \cap C \neq \{\}$, then there is some $t \in C$ such that $B \cup \{t\} \subseteq B'$. Because $B \cup \{t\}$ is inconsistent, no branch that is a superset of $B \cup \{t\}$ is evident. Thus $B'$ is not evident. □

Consider some clashed tableau. By Lemma 2.7, all its branches are inconsistent. By induction from the leaves to the root using Lemma 2.8, the initial formula is inconsistent.

## 2.3 Termination

**Definition 2.9.** The **step relation** is the smallest relation $(\to)$ such that for every step $\frac{B}{C} \in \hat{\mathcal{T}}$ and for every $t \in C$, we have that $B \to (B \cup \{t\})$. A tableau system $\mathcal{T}$ is called **terminating** if $(\to)$ is terminating.

Termination guarantees that an attempt to build a tableau cannot diverge because a path in a tableau cannot be extended infinitely often. So, if we have a terminating tableau system, then we can decide the consistency of a formula because after finitely many steps, we always obtain a tableau which cannot be extended any longer. The initial formula is consistent iff this tableau contains an evident branch. If consistency of formulas coincides with satisfiability in a given logic, then the tableau method gives us a decision procedure for this logic.

For example, the tableau system for propositional logic from Figure 1.1 is terminating. Every branch obtained by extending $B_1$ contains only subformulas of formulas in $B_1$. Thus the cardinality of these branches is bounded. Since the cardinality of a branch strictly increases for every step of $(\to)$, this gives an upper bound for the length of a path $B_1 \to B_2 \to \cdots \to B_n$.

In the following, we assume that $\mathcal{T}$ is terminating.

# 3 Correctness of Depth-First Search

Next, we will define depth-first search for tableaux. We describe DFS as the recursive procedure in Figure 3.1. The syntax is ML-like. DFS($B$) yields whether the branch $B$ is consistent. The algorithm "walks" through the tableau tree, where the argument $B$ always contains the current branch and every recursive call corresponds to going one step down in the tree. It can be read as follows.

- If $B$ is evident, then return "true".
- Else select some step to be applied. We assume this selection to be deterministic but do not specify it. An implementation might apply some intelligent heuristics at this point.
- If a clashing step $\frac{B}{\{\}}$ has been selected, then return "false".
- If a step with a single conclusion $\frac{B}{\{s\}}$ has been selected, then continue with $B \cup \{s\}$ by calling DFS($B \cup \{s\}$). This corresponds to a node in the tableau that has a single child.
- If a branching step $\frac{B}{\{s,t\}}$ has been selected, then first call DFS($B \cup \{s\}$). If this yields "true", then return "true". If this yields "false", then call DFS($B \cup \{t\}$).
  This corresponds to a branching point in the tableau. We first try to extend $B$ with $s$. This corresponds to the left subtree of the branching point. We always first fully explore the left subtree before considering the right subtree.

$$\text{DFS} : \{B \mid B \text{ is a branch}\} \to \{\text{true}, \text{false}\}$$

$$
\begin{aligned}
\text{DFS}(B) = \quad & \textsc{if}\ B \text{ evident} \\
& \textsc{then}\ \text{true} \\
& \textsc{else} \\
& \quad \textsc{choose some}\ \tfrac{B}{C} \in \hat{\mathcal{T}} \\
& \qquad \textsc{case}\ C\ \textsc{of} \\
& \qquad\quad \{\} : \quad \text{false} \\
& \qquad\quad \{s\} : \quad \text{DFS}(B \cup \{s\}) \\
& \qquad\quad \{s,t\} : \quad \text{DFS}(B \cup \{s\})\ \textsc{or}\ \text{DFS}(B \cup \{t\})
\end{aligned}
$$

Figure 3.1: The DFS-algorithm

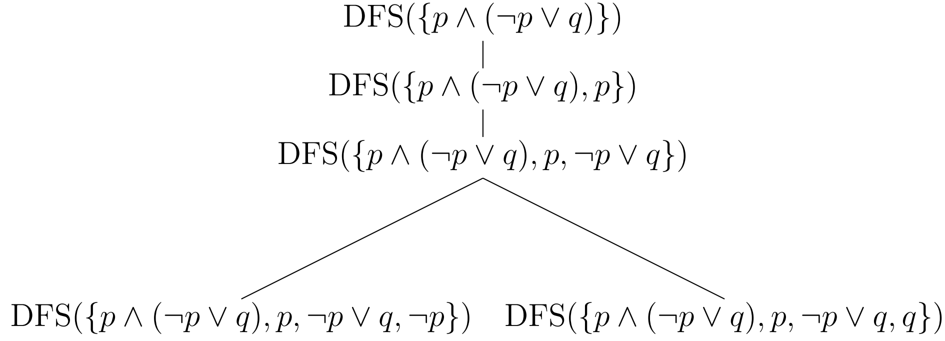


Figure 3.2: Recursion tree of DFS($\{p \wedge (\neg p \vee q)\}$)

If DFS($B \cup \{s\}$) = true, then we know that also $B$ is consistent and we return true. If DFS($B \cup \{s\}$) = false, then we know that $B \cup \{s\}$ is inconsistent. Thus we try $B \cup \{t\}$. This corresponds to the right subtree of the branching point.

As an example, the recursion tree of DFS($\{\{p \wedge (\neg p \vee q)\}\}$) is depicted in Figure 3.2. Every node corresponds to one procedure call. The paths in the tree from left to right correspond to the call stacks during the execution. Note that the recursion tree in Figure 3.2 has the same structure as the tableau tree in Figure 1.2. The procedure call DFS($\{p \wedge (\neg p \vee q), p, \neg p \vee q, \neg p\}$) yields "false" because the branch $\{p \wedge (\neg p \vee q), p, \neg p \vee q, \neg p\}$, which corresponds to the left branch in Figure 1.2, is clashed. Thus the procedure tries the other alternative of the branching point and calls DFS($\{p \wedge (\neg p \vee q), p, \neg p \vee q, q\}$). This call yields "true" because the argument is an evident branch. This value propagates upwards and hence DFS($\{p \wedge (\neg p \vee q)\}$) = true.

One easily verifies that the result of the procedure is always well-defined. One can always choose a step if the branch is not evident and the case distinction is exhaustive because $|C| \leq 2$.

Since we assume that the tableau system terminates, we can show termination of the procedure. From the terminating step relation $(\rightarrow)$, we obtain a well-founded order containing the terminating relation by taking the transitive closure $(>) := (\rightarrow^*)$. We call $(>)$ the **step order**. Since we have $B > (B \cup \{s\})$ and $B > (B \cup \{t\})$ for all $s, t \in \mathcal{F}$ s.t. $\frac{B}{\{s\}} \in \hat{\mathcal{T}}$ or $\frac{B}{\{s,t\}} \in \hat{\mathcal{T}}$, the recursive calls are always done on strictly smaller arguments. Hence the procedure terminates.

Now we can show the soundness and completeness of DFS in following sense.

**Theorem 3.1.** *Let $\mathcal{T}$ be a terminating tableau system and let* DFS *be the procedure defined in Figure 3.1. Then for all branches $B$, we have* DFS($B$) = *true iff $B$ is consistent.*

*Proof.* We show the statements by induction on the size of the argument with respect to the step order $(>)$.

"$\Rightarrow$": Let DFS($B$) = true. We show that $B$ is consistent. If $B$ is evident, then it is consistent. Else the procedure chooses some step $\frac{B}{C} \in \hat{\mathcal{T}}$. We make a case distinction on $C$.

- It is impossible that $C = \{\}$ because this would yield DFS($B$) = false contradicting DFS($B$) = true.
- If $C = \{s\}$, then DFS($B$) = DFS($B \cup \{s\}$) and hence by assumption DFS($B \cup \{s\}$) = true. By the inductive hypothesis, this means that $B \cup \{s\}$ is consistent. By Proposition 2.6, it follows that $B$ is consistent.
- If $C = \{s, t\}$, then DFS($B \cup \{s\}$) = true or DFS($B \cup \{t\}$) = true. By the inductive hypothesis, this means that $B \cup \{s\}$ or $B \cup \{t\}$ is consistent. In both cases, it follows that $B$ is consistent.

"$\Leftarrow$": Let DFS($B$) = false. We show that $B$ is inconsistent. By assumption, $B$ cannot be evident because this would yield DFS($B$) = true. Thus the procedure chooses some step $\frac{B}{C} \in \hat{\mathcal{T}}$. We again make a case distinction on $C$.

- If $C = \{\}$, then $\frac{B}{\{\}} \in \hat{\mathcal{T}}$. Thus $B$ is clashed and hence inconsistent.
- If $C = \{s\}$, then we have $\frac{B}{\{s\}} \in \hat{\mathcal{T}}$ and DFS($B \cup \{s\}$) = false. By the inductive hypothesis, this means that $B \cup \{s\}$ is inconsistent. By Lemma 2.8, it follows that $B$ is inconsistent.
- If $C = \{s, t\}$, then we have $\frac{B}{\{s,t\}} \in \hat{\mathcal{T}}$, DFS($B \cup \{s\}$) = false and DFS($B \cup \{t\}$) = false. By the inductive hypothesis, this means that $B \cup \{s\}$ and $B \cup \{t\}$ are inconsistent. By Lemma 2.8, it follows that $B$ is inconsistent. □

# 4 Correctness of Backjumping

In this chapter, we will formalize and prove correct DFS with backjumping.

## 4.1 Dependency Sequents

To prove backjumping correct, we give semantics to dependency sets. For this, from now on, we use dependency sets $A \subseteq \mathcal{F}$, which consist of formulas instead of numbers. A dependency sequent $A \Rightarrow s$ is a pair of a formula $s$ and its dependency set $A$. The dependency sequent of a clash is written as $A \Rightarrow \otimes$. Backjumping will only use valid dependency sequents.

**Definition 4.1.** A **dependency sequent** is a pair $A \Rightarrow s$ or $A \Rightarrow \otimes$ where $s \in \mathcal{F}$, $A \subseteq \mathcal{F}$ and $A$ is finite. A dependency sequent $A \Rightarrow s$ is called **valid** if for every evident branch $B \supseteq A$, we have $s \in B$. A sequent $A \Rightarrow \otimes$ is **valid** if there is no evident branch $B \supseteq A$.

We use the usual notation for sequents and write

- $s_1, \ldots, s_n \Rightarrow t$ for $\{s_1, \ldots, s_n\} \Rightarrow t$
- $A, A' \Rightarrow s$ for $(A \cup A') \Rightarrow s$
- $A, s \Rightarrow t$ for $(A \cup \{s\}) \Rightarrow t$

**Proposition 4.2.** *The dependency sequent $A \Rightarrow \otimes$ is valid iff $A$ is inconsistent.*

Therefore, to prove that $s$ is inconsistent, it suffices to show that the sequent $s \Rightarrow \otimes$ is valid. This is what backjumping will do.

### 4.1.1 Example

We will now show how dependency sequents can be used to prove that the branches $c$ and $d$ in Figure 1.5 are unnecessary. This also constitutes an example of a tableau search with backjumping, which we will define later. We take the tableau from Figure 1.6 and derive a valid dependency sequent for every formula on an explored branch. This is done in the same order as before (depth-first). We require that a formula on a branch $B$ is labeled with a dependency set $A \subseteq B$. The resulting tableau is depicted in Figure 4.1.

$$\begin{array}{ccc} s & \Rightarrow & s \\ s & \Rightarrow & q_1 \vee \neg q_2 \\ s & \Rightarrow & q_2 \\ s & \Rightarrow & p_1 \vee p_2 \\ s & \Rightarrow & \neg q_1 \vee \neg q_2 \end{array}$$

1

| | | | |
|---|---|---|---|
| 2: $q_1 \Rightarrow q_1$ | | | $s \Rightarrow \neg q_2$<br>$s \Rightarrow \otimes$ |
| 3: $p_1 \Rightarrow p_1$ | | $p_2$ | |
| $\neg q_1 \Rightarrow \neg q_1$<br>$q_1, \neg q_1 \Rightarrow \otimes$ | $s, q_1 \Rightarrow \neg q_2$<br>$s, q_1 \Rightarrow \otimes$ | | |
| $a$ | $b$ | $c$ | $d$ |

where $s := (q_1 \vee \neg q_2) \wedge q_2 \wedge (p_1 \vee p_2) \wedge (\neg q_1 \vee \neg q_2)$

Figure 4.1: A tableau refutation with backjumping

Note how the numbers in the dependency sets in Figure 1.6 correspond to the formulas in the dependency sets in Figure 4.1. The initial formula $s$ corresponds to the number 0 and the left child of a branching point replaces the index of this branching point.

The initial formula $(q_1 \vee \neg q_2) \wedge q_2 \wedge (p_1 \vee p_2) \wedge (\neg q_1 \vee \neg q_2)$, abbreviated as $s$, is assigned the dependency set $\{s\}$. This is possible because $s \Rightarrow s$ is trivially valid. The formula $q_1 \vee \neg q_2$ is derived from $s$. Because of the rule $\frac{\{s\}}{\{q_1 \vee \neg q_2\}}$, every evident branch $B$ that contains $s$ also contains $q_1 \vee \neg q_2$. So $s \Rightarrow q_1 \vee \neg q_2$ is a valid dependency sequent. Analogously, the formulas $q_2$, $p_1 \vee p_2$ and $\neg q_1 \vee \neg q_2$ have the same dependency set. The formula $q_1$ is assigned the dependency set $\{q_1\}$ because it is the left child of a branching point. Again, $q_1 \Rightarrow q_1$ is trivially valid. Analogously for $p_1$ and $\neg q_1$.

The clash in branch $a$ is labeled with the dependency set $\{q_1, \neg q_1\} = \{q_1\} \cup \{\neg q_1\}$, because it is derived from the formulas $q_1$ and $\neg q_1$ that have the dependency sets $\{q_1\}$ and $\{\neg q_1\}$, respectively. This yields the valid sequent $q_1, \neg q_1 \Rightarrow \otimes$ because there is no evident branch $B$ containing both $q_1$ and $\neg q_1$.

Because $\neg q_1$, the left child of branching point 3, is contained in the dependency set of the clash, branching point 3 is responsible for the clash. Thus we backtrack to branching point 3 and continue with branch $b$. We have now to find a valid dependency set $A$ for $\neg q_2$. We derive it from the dependency set of the clash in branch $a$ and from the dependency set of the disjunction $\neg q_1 \vee \neg q_2$ that caused the branching point. Consider an evident branch $B \supseteq A$. We take the dependency set of $\neg q_1 \vee \neg q_2$, that is $\{s\}$, to be in $A$. Thus we know that either $\neg q_1$ or $\neg q_2$ is in $B$. Additionally, we take the dependency set of the clash $\{q_1, \neg q_1\}$ except for $\neg q_1$, that is $\{q_1\}$, to be in $A$. Thus we know that $B$ does not contain $\neg q_1$. Altogether, we know that $\neg q_2$ must be in $B$. Hence we can take $A = \{s\} \cup (\{q_1, \neg q_1\} - \{\neg q_1\}) =$

$$
\begin{array}{ccc}
s & \Rightarrow & s \\
s & \Rightarrow & p \vee q_1 \\
s & \Rightarrow & \neg q_1 \vee \neg q_2 \\
s & \Rightarrow & q_1 \\
s & \Rightarrow & q_2
\end{array}
$$

1 ——————————————————————

$$
\begin{array}{c|c}
\begin{array}{c}
p \Rightarrow p \\
\hline
\begin{array}{ccc|ccc}
\neg q_1 & \Rightarrow & \neg q_1 & s & \Rightarrow & \neg q_2 \\
s, \neg q_1 & \Rightarrow & \otimes & s & \Rightarrow & \otimes
\end{array}
\end{array}
& q_1
\end{array}
$$

2 (branching point between $\neg q_1$ and $s \Rightarrow \neg q_2$)

$a \qquad b \qquad c$

where $s := (p \vee q_1) \wedge (\neg q_1 \vee \neg q_2) \wedge q_1 \wedge q_2$

Figure 4.2: Another tableau with backjumping

$\{s, q_1\}$ yielding the valid dependency sequent $s, q_1 \Rightarrow \neg q_2$.

The dependency set of the clash of branch $b$ is then derived from the dependency sets of $q_2$ and $\neg q_2$, which are $\{s\}$ and $\{s, q_1\}$, respectively. So we obtain the valid dependency sequent $s, q_1 \Rightarrow \otimes$. The dependency set $\{s, q_1\}$ is then used for backtracking. We know that every branch containing $\{s, q_1\}$ is inconsistent. We do not need to explore any of them. So we have to backtrack until one element of the dependency set of the clash is no longer on the current branch. Thus we can safely backtrack to branching point 1 continuing with branch $d$ because the other branches all contain $\{s, q_1\}$. Hence we save the work of exploring branch $c$, which corresponds to the branches $c$ and $d$ in Figure 1.5. In branch $d$, we derive the sequent $s \Rightarrow \neg q_2$ with the same argument as for $\neg q_2$ in branch $b$. Then we can conclude that the sequent $s \Rightarrow \otimes$ is valid. This means that $s$ is inconsistent.

We realize backjumping by deriving valid dependency sequents while computing a tableau. Having these sequents at hand, we can avoid exploring parts of the tableau. We do not miss evident branches since all the branches that we skip are provably inconsistent. We will show that if backjumping terminates without finding an evident branch, then the sequent $s \Rightarrow \otimes$ is valid and hence $s$ is inconsistent.

To determine the branching point to which we backtrack, it suffices to know the most recent formula in the dependency set of a clash, i.e., the formula referencing the most recent branching point. Why do we then store the whole dependency set? Let us look at the example in Figure 4.2. If we just knew that the dependency set of the first clash contains $\neg q_1$, then we would have to give $\neg q_2$ the dependency set $\{s, p\}$ since we would not know whether the dependency set of the first clash contained $p$ or not. Then the clash in branch $b$ would have the dependency set $\{s, p\}$ forcing us to explore branch $c$. Hence backjumping would not have any effect.

$$D_0 \frac{}{s \Rightarrow s}$$

$$D_\alpha \frac{A_1 \Rightarrow s_1 \quad \cdots \quad A_n \Rightarrow s_n}{A_1, \ldots, A_n \Rightarrow t} \; \tfrac{\{s_1,\ldots,s_n\}}{\{t\}} \in \mathcal{T}$$

$$D_\beta \frac{A_1 \Rightarrow s_1 \quad \cdots \quad A_n \Rightarrow s_n \quad A', t_1 \Rightarrow \otimes}{A_1, \ldots, A_n, A' \Rightarrow t_2} \; \tfrac{\{s_1,\ldots,s_n\}}{\{t_1,t_2\}} \in \mathcal{T}$$

$$D_\otimes \frac{A_1 \Rightarrow s_1 \quad \cdots \quad A_n \Rightarrow s_n}{A_1, \ldots, A_n \Rightarrow \otimes} \; \tfrac{\{s_1,\ldots,s_n\}}{\{\}} \in \mathcal{T}$$

Figure 4.3: Derivation rules for valid dependency sequents

### 4.1.2 Relation to Propositional Logic

Note that for the propositional tableau system, the sequent $s_1, \ldots, s_n \Rightarrow t$ is valid iff the propositional formula $(s_1 \wedge \cdots \wedge s_n) \rightarrow t$ is valid, where "$\rightarrow$" stands for logical implication. This formula is equivalent to the formula $\neg s_1 \vee \cdots \vee \neg s_n \vee t$. Analogously, a clash sequent $s_1, \ldots, s_n \Rightarrow \otimes$ corresponds to the formula $\neg s_1 \vee \cdots \vee \neg s_n$.

For DPLL, Nieuwenhuis et al. [25] use a propositional disjunction of literals $\neg l_1 \vee \cdots \vee \neg l_n$ to express the backjumping information about a clash. The literals $l_i$ are possibly negated propositional variables that stand for nondeterministic decisions. These disjunctions are called "backjump clauses" in this paper and correspond to a clash sequent $l_1, \ldots, l_n \Rightarrow \otimes$ in our system. The other sequents do not have counterparts in [25] because backjump clauses are calculated from scratch after a clash has been detected.

### 4.1.3 Dependency Sequent Rules

The reasoning we used to justify the validity of the sequents in Figure 4.1 can be generalized. This is done by the rules in Figure 4.3, which derive valid dependency sequents from valid dependency sequents. The rule $D_\alpha$ corresponds to the application of a tableau rule with a single conclusion. $D_0$ is used for the left conclusion of a branching tableau rule and $D_\beta$ for its right conclusion. The rule $D_\otimes$ corresponds to the application of a clashing tableau rule. The instantiation of these rules for the propositional tableau system is depicted in Figure 4.4.

Figure 4.5 shows how the rules in Figure 4.4 can be used to derive the sequents in Figure 4.1.

**Lemma 4.3.** *The rules in Figure 4.3 derive valid dependency sequents.*

$$D_0^{\text{prop}} \frac{}{s \Rightarrow s}$$

$$D_\alpha^{\text{prop}} \frac{A \Rightarrow s_1 \wedge \cdots \wedge s_n}{A \Rightarrow s_i}\ {}_{i \in [1,n]}$$

$$D_\beta^{\text{prop}} \frac{A \Rightarrow s \vee t \quad A', s \Rightarrow \otimes}{A, A' \Rightarrow t}$$

$$D_\otimes^{\text{prop}} \frac{A \Rightarrow s \quad A' \Rightarrow \neg s}{A, A' \Rightarrow \otimes}$$

Figure 4.4: Instantiation of the rules in Figure 4.3 for the propositional tableau system

*Proof.* For every rule in Figure 4.3, we show that the derived dependency sequent is valid under the assumption that the sequents in the premise are valid.

$D_0$ Validity of the sequent $s \Rightarrow s$ is immediate by definition.

$D_\alpha$ $A_1, \ldots, A_n \Rightarrow t$ is valid iff every evident branch $B \supseteq A_1 \cup \cdots \cup A_n$ contains $t$. Let $B$ be such a branch. It suffices to show that $t \in B$. Because the sequents in the premise are valid, $\{s_1, \ldots, s_n\} \subseteq B$. Since $\{s_1, \ldots, s_n\} \subseteq B$ and $\frac{\{s_1,\ldots,s_n\}}{\{t\}} \in \mathcal{T}$, we have that $t \in B$ by Lemma 2.4.

$D_\beta$ $A_1, \ldots, A_n \Rightarrow t_2$ is valid iff every evident branch $B \supseteq A_1 \cup \cdots \cup A_n \cup A'$ contains $t_2$. Let $B$ be such a branch. It suffices to show that $t_2 \in B$. By assumption, $\{s_1, \ldots, s_n\} \subseteq B$. Since $\{s_1, \ldots, s_n\} \subseteq B$ and $\frac{\{s_1,\ldots,s_n\}}{\{t_1,t_2\}} \in \mathcal{T}$, we have that $B \cap \{t_1, t_2\} \neq \{\}$ by Lemma 2.4. Thus either $t_1 \in B$ or $t_2 \in B$. Because $A', t_1 \Rightarrow \otimes$ is valid, there is no evident branch containing $A'$ and $t_1$. Since $B$ contains $A'$, it cannot contain $t_1$ and therefore $t_2 \in B$.

$D_\otimes$ $A_1, \ldots, A_n \Rightarrow \otimes$ is valid iff there is no evident branch $B \supseteq A_1 \cup \cdots \cup A_n$. Suppose, for contradiction, that such a branch $B$ exists. By assumption, $\{s_1, \ldots, s_n\} \subseteq B$. Since $\{s_1, \ldots, s_n\} \subseteq B$ and $\frac{\{s_1,\ldots,s_n\}}{\{\}} \in \mathcal{T}$, we have $B \cap \{\} \neq \{\}$ by Lemma 2.4. Contradiction. □

## 4.2 DFS with Backjumping

We will now formulate DFS with backjumping, which we just call backjumping from now on. We present backjumping as a recursive procedure as we have done for DFS. It is shown in Figure 4.6.

| # | sequent | sequent rule |
|---|---|---|
| 1 | $s \Rightarrow s$ | $D_0^{\text{prop}}$ |
| 2 | $s \Rightarrow q_1 \vee \neg q_2$ | $D_\alpha^{\text{prop}}(1)$ |
| 3 | $s \Rightarrow q_2$ | $D_\alpha^{\text{prop}}(1)$ |
| 4 | $s \Rightarrow p_1 \vee p_2$ | $D_\alpha^{\text{prop}}(1)$ |
| 5 | $s \Rightarrow \neg q_1 \vee \neg q_2$ | $D_\alpha^{\text{prop}}(1)$ |
| 6 | $q_1 \Rightarrow q_1$ | $D_0^{\text{prop}}$ |
| 7 | $p_1 \Rightarrow p_1$ | $D_0^{\text{prop}}$ |
| 8 | $\neg q_1 \Rightarrow \neg q_1$ | $D_0^{\text{prop}}$ |
| 9 | $q_1, \neg q_1 \Rightarrow \otimes$ | $D_\otimes^{\text{prop}}(6, 8)$ |
| 10 | $s, q_1 \Rightarrow \neg q_2$ | $D_\beta^{\text{prop}}(5, 9)$ |
| 11 | $s, q_1 \Rightarrow \otimes$ | $D_\otimes^{\text{prop}}(3, 10)$ |
| 12 | $s \Rightarrow \neg q_2$ | $D_\beta^{\text{prop}}(2, 11)$ |
| 13 | $s \Rightarrow \otimes$ | $D_\otimes^{\text{prop}}(3, 12)$ |

Figure 4.5: Derivation of the sequents from Figure 4.1

$$\text{BJ} : \{\Sigma \mid \Sigma \text{ is an annotated branch}\} \rightarrow \{A \Rightarrow \otimes \mid A \subseteq \mathcal{F}\} \cup \{\text{true}\}$$

$$
\begin{array}{ll}
\text{BJ}(\Sigma) = & \textsc{if } \mathcal{B}(\Sigma) \text{ evident} \\
& \textsc{then } \text{true} \\
& \textsc{else} \\
& \quad \textsc{choose some } \frac{\{s_1,\ldots,s_n\}}{C} \in \mathcal{T} \text{ that applies to } \mathcal{B}(\Sigma) \\
& \quad \textsc{choose } A_1,\ldots,A_n \text{ s.t. } \{A_1 \Rightarrow s_1,\ldots,A_n \Rightarrow s_n\} \subseteq \Sigma \\
& \quad \textsc{case } C \textsc{ of} \\
& \qquad \{\} : \\
& \qquad\quad A_1,\ldots,A_n \Rightarrow \otimes \\
& \qquad \{t\} : \\
& \qquad\quad \text{BJ}(\Sigma \cup \{(A_1,\ldots,A_n \Rightarrow t)\}) \\
& \qquad \{t_1, t_2\} : \\
& \qquad\quad \textsc{case } \text{BJ}(\Sigma \cup \{t_1 \Rightarrow t_1\}) \textsc{ of} \\
& \qquad\qquad \text{true} : \\
& \qquad\qquad\quad \text{true} \\
& \qquad\qquad (A' \Rightarrow \otimes) : \\
& \qquad\qquad\quad \textsc{if } t_1 \notin A' \\
& \qquad\qquad\quad \textsc{then } (A' \Rightarrow \otimes) \\
& \qquad\qquad\quad \textsc{else } \text{BJ}(\Sigma \cup \{(A_1,\ldots,A_n,(A' - \{t_1\}) \Rightarrow t_2)\})
\end{array}
$$

Figure 4.6: The backjumping procedure

The procedure operates on sets of dependency sequents instead of formulas. We use the following notions.

**Definition 4.4.** We call a finite set $\Sigma \subseteq \{A \Rightarrow s \mid A \subseteq \mathcal{F}, A \text{ is finite}, s \in \mathcal{F}\}$ an **annotated branch**. We call an annotated branch $\Sigma$ **valid** if all dependency sequents in $\Sigma$ are valid.

For every annotated branch $\Sigma$, we define the corresponding branch $\mathcal{B}(\Sigma) := \{s \mid (A \Rightarrow s) \in \Sigma\}$ and the set of dependencies $\mathcal{D}(\Sigma) := \bigcup\{A \mid (A \Rightarrow s) \in \Sigma\}$.

We apply the procedure only to valid annotated branches $\Sigma$. As we will see, the procedure maintains the invariant that the argument $\Sigma$ is a valid annotated branch, that is, all sequents in $\Sigma$ are valid.

As for the DFS-procedure, one easily verifies that the procedure is well-defined. The first choose-construct always finds a rule because if $\mathcal{B}(\Sigma)$ is not evident, then there is some rule that applies to $\mathcal{B}(\Sigma)$. The second choose-construct always finds $A_1, \ldots, A_n$ because $\{s_1, \ldots, s_n\} \subseteq \mathcal{B}(\Sigma)$ and hence, by the definition of $\mathcal{B}$, there have to be sequents for $s_1, \ldots, s_n$ in $\Sigma$. Again, we assume these selections to be deterministic. One can even show that the procedure maintains the invariant that for every formula $s \in \mathcal{B}(\Sigma)$, there is exactly one $(A \Rightarrow s) \in \Sigma$. Thus the second choose-construct is deterministic by construction provided that one applies BJ only to annotated branches satisfying this invariant.

The termination argument stays the same, too. If $\mathrm{BJ}(\Sigma)$ can call $\mathrm{BJ}(\Sigma')$, then $\mathcal{B}(\Sigma) > \mathcal{B}(\Sigma')$ for the step order $(>)$, which is well-founded. Thus BJ always terminates.

$\mathrm{BJ}(\Sigma)$ either yields that $\mathcal{B}(\Sigma)$ is a consistent branch or it yields a valid clash sequent whose dependency set is a subset of $\mathcal{D}(\Sigma)$.

**Lemma 4.5.** *Let $\Sigma$ be some valid annotated branch. If $\mathrm{BJ}(\Sigma) = true$, then $\mathcal{B}(\Sigma)$ is consistent. If $\mathrm{BJ}(\Sigma) = (A \Rightarrow \otimes)$, then $A \Rightarrow \otimes$ is a valid dependency sequent and $A \subseteq \mathcal{D}(\Sigma)$.*

*Proof.* We show the statements by induction on the size of the argument with respect to the step order $(>)$. We use the rules in Figure 4.3 to argue the validity of sequents.

Let $\mathrm{BJ}(\Sigma) = \text{true}$. We show that $\mathcal{B}(\Sigma)$ is consistent. If $\mathcal{B}(\Sigma)$ is evident, then $\mathcal{B}(\Sigma)$ is consistent. Else the algorithm chooses some rule $\frac{\{s_1,\ldots,s_n\}}{C} \in \mathcal{T}$ that applies to $\mathcal{B}(\Sigma)$ and $A_1, \ldots, A_n$ such that $\{A_1 \Rightarrow s_1, \ldots, A_n \Rightarrow s_n\} \subseteq \Sigma$. We make a case distinction on $C$.

- It is impossible that $C = \{\}$ because this would yield $\mathrm{BJ}(\Sigma) = (A_1, \ldots, A_n \Rightarrow \otimes)$ contradicting $\mathrm{BJ}(\Sigma) = \text{true}$.

- Let $C = \{t\}$. By $D_\alpha$, we have that $A_1, \ldots, A_n \Rightarrow t$ is valid. Thus $\Sigma \cup \{(A_1, \ldots, A_n \Rightarrow t)\}$ is a valid annotated branch.
  We have $\mathrm{BJ}(\Sigma \cup \{(A_1, \ldots, A_n \Rightarrow t)\}) = \mathrm{BJ}(\Sigma) = \text{true}$. By the inductive hypothesis, this means that $\mathcal{B}(\Sigma \cup \{(A_1, \ldots, A_n \Rightarrow t)\}) = \mathcal{B}(\Sigma) \cup \{t\}$ is consistent. By Proposition 2.6, it follows that $\mathcal{B}(\Sigma)$ is consistent.
- Let $C = \{t_1, t_2\}$. By $D_0$, we have that $t_1 \Rightarrow t_1$ is valid. Thus $\Sigma \cup \{t_1 \Rightarrow t_1\}$ is a valid annotated branch.
  If $\mathrm{BJ}(\Sigma \cup \{t_1 \Rightarrow t_1\}) = \text{true}$, then by the inductive hypothesis, $\mathcal{B}(\Sigma) \cup \{t_1\}$ is consistent. Thus $\mathcal{B}(\Sigma)$ is consistent.
  Else $\mathrm{BJ}(\Sigma \cup \{t_1 \Rightarrow t_1\}) = (A' \Rightarrow \otimes)$ and $\mathrm{BJ}(\Sigma \cup \{(A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2)\}) = \text{true}$. By the inductive hypothesis, $A' \Rightarrow \otimes$ is valid. Using this, by $D_\beta$, we have that $A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2$ is valid. Hence $\Sigma \cup \{(A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2)\}$ is a valid annotated branch. Using the inductive hypothesis on $\mathrm{BJ}(\Sigma \cup \{(A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2)\}) = \text{true}$, we obtain that $\mathcal{B}(\Sigma) \cup \{t_2\}$ is consistent and hence $\mathcal{B}(\Sigma)$ is consistent.

Let $\mathrm{BJ}(\Sigma) = (A'' \Rightarrow \otimes)$. We show that $A'' \Rightarrow \otimes$ is a valid dependency sequent and $A'' \subseteq \mathcal{D}(\Sigma)$. It is impossible that $\mathcal{B}(\Sigma)$ is consistent because this would yield $\mathrm{BJ}(\Sigma) = \text{true}$ contradicting $\mathrm{BJ}(\Sigma) = (A'' \Rightarrow \otimes)$. Thus the algorithm chooses some rule $\frac{\{s_1, \ldots, s_n\}}{C} \in \mathcal{T}$ that applies to $\mathcal{B}(\Sigma)$ and $A_1, \ldots, A_n$ such that $\{A_1 \Rightarrow s_1, \ldots, A_n \Rightarrow s_n\} \subseteq \Sigma$. We make a case distinction on $C$.

- Let $C = \{\}$. We have $\frac{\{s_1, \ldots, s_n\}}{\{\}} \in \mathcal{T}$ and $A'' = A_1, \ldots, A_n$. By $D_\otimes$, we have that $A_1, \ldots, A_n \Rightarrow \otimes$ is valid. $A_1 \cup \cdots \cup A_n \subseteq \mathcal{D}(\Sigma)$ holds by construction. Thus $A'' \Rightarrow \otimes$ is as requested.
- Let $C = \{t\}$. We have $\mathrm{BJ}(\Sigma \cup \{(A_1, \ldots, A_n \Rightarrow t)\}) = (A'' \Rightarrow \otimes)$. By $D_\alpha$, $A_1, \ldots, A_n \Rightarrow t$ is valid. Therefore, by the inductive hypothesis, we have that $A'' \Rightarrow \otimes$ is a valid sequent with $A'' \subseteq \mathcal{D}(\Sigma \cup \{(A_1, \ldots, A_n \Rightarrow t)\}) = \mathcal{D}(\Sigma) \cup A_1 \cup \cdots \cup A_n = \mathcal{D}(\Sigma)$ since $A_1 \cup \cdots \cup A_n \subseteq \mathcal{D}(\Sigma)$.
- Let $C = \{t_1, t_2\}$. We have $\mathrm{BJ}(\Sigma \cup \{t_1 \Rightarrow t_1\}) = (A' \Rightarrow \otimes)$. By $D_0$, $t_1 \Rightarrow t_1$ is valid. Thus, by the inductive hypothesis, we have that $A' \Rightarrow \otimes$ is a valid sequent with $A' \subseteq \mathcal{D}(\Sigma \cup \{t_1 \Rightarrow t_1\}) = \mathcal{D}(\Sigma) \cup \{t_1\}$.
  If $t_1 \notin A'$, then $A' \subseteq \mathcal{D}(\Sigma)$ and $A'' = A'$. Thus $A'' \Rightarrow \otimes$ is as requested.
  If $t_1 \in A'$, then $\mathrm{BJ}(\Sigma \cup \{(A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2)\}) = (A'' \Rightarrow \otimes)$. By $D_\beta$, we have that $A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2$ is valid. Again, by the inductive hypothesis, we have that $A'' \Rightarrow \otimes$ is a valid dependency sequent with $A'' \subseteq \mathcal{D}(\Sigma \cup \{(A_1, \ldots, A_n, (A' - \{t_1\}) \Rightarrow t_2)\}) = \mathcal{D}(\Sigma) \cup (A' - \{t_1\})$. Since $A' \subseteq \mathcal{D}(\Sigma) \cup \{t_1\}$, we have $A'' \subseteq \mathcal{D}(\Sigma)$. □

**Theorem 4.6.** *Let $\mathcal{T}$ be a terminating tableau system and let* BJ *be the procedure defined in Figure 4.6. Then for all branches $B = \{s_1, \ldots, s_n\}$, we have* $\mathrm{BJ}(\{s_1 \Rightarrow s_1, \ldots, s_n \Rightarrow s_n\}) = \mathit{true}$ *iff $B$ is consistent.*

*Proof.* BJ always yields a result because it is terminating.

If $\mathrm{BJ}(\{s_1 \Rightarrow s_1, \ldots, s_n \Rightarrow s_n\}) = \text{true}$, then $B$ is consistent by Lemma 4.5.

Else $\mathrm{BJ}(\{s_1 \Rightarrow s_1, \ldots, s_n \Rightarrow s_n\}) = (A \Rightarrow \otimes)$. We have to show that $B$ is inconsistent. The sequent $A \Rightarrow \otimes$ is valid and $A \subseteq \mathcal{B}(\{s_1 \Rightarrow s_1, \ldots, s_n \Rightarrow s_n\}) = B$, again by Lemma 4.5. Thus $B$ is inconsistent. □

Note that the rules for dependency sequents in Figure 4.3 can be seen as an independent calculus. Since the backjumping procedure only derives sequents according to these rules, one can derive potentially more sequents if one uses the rules freely like in resolution. Since $\mathrm{BJ}(\{s \Rightarrow s\})$ can derive $\{s\} \Rightarrow \otimes$ for every inconsistent formula $s$, we also get a completeness result for the free calculus provided that the tableau system terminates. Obviously, the shortest possible derivation of $s \Rightarrow \otimes$ in the free system is at most as long as the shortest tableau-proof with backjumping. So this way, one can transform any tableau system fitting in our framework to a resolution-like proof system. However, a further investigation lies out of the scope of this thesis.

## 4.3 Learning

Nieuwenhuis et al. [25] also suggest to use backjump clauses for clause learning. This means that after a backjump clause has been computed, it is added to the set of clauses of the original problem and used in the following computations. Clause learning is a standard optimization technique for SAT-solvers [24].

Thus, it might be beneficial to learn clash sequents globally. In our setting, this could be realized as follows. While constructing a tableau, we store a global set $L$ of learned valid clash sequents. It collects clash sequents from all branches of the tableau. If one has learned a clash sequent $A \Rightarrow \otimes$, i.e., if $(A \Rightarrow \otimes) \in L$, then one knows that all branches that contain $A$ are inconsistent. Therefore, one can stop exploring a branch as soon as it contains $A$. This can be expressed with the following tableau rule.

$$\frac{A}{\quad}\ (A \Rightarrow \otimes) \in L$$

For every set of valid clash sequents $L$, this is a sound rule. The dependency set of a clash caused by this rule is the union of the dependency sets of the formulas in $A$.

# 5 The Modal Case

In the following, we will present a tableau system for the basic modal logic $K$ that fits in our framework and can thus be extended with backjumping. It is similar to the prefixed tableau systems developed by Fitting [7] and Massacci [23]. These systems use freshness conditions (see Section 5.2) that cannot be expressed directly in our framework. So we needed to devise a tableau system that does not need freshness conditions.

## 5.1 Syntax and Semantics of K

The **basic modal logic K** augments propositional logic with the two modal operators $\Diamond$ and $\Box$. The description logic $\mathcal{ALC}$ is a syntactic variant of K. The complexity of the satisfiability problem is PSPACE-complete [21].

We define syntax and semantics of K based on Kaminski and Smolka [20]. The syntax of formulas is given by the grammar

$$\begin{aligned} p, q &::= \text{predicate} \\ s, t, u &::= p \mid \neg p \mid s \wedge s \mid s \vee s \mid \Diamond s \mid \Box s \end{aligned}$$

where $p$ and $q$ range over a countable set of names, called predicates. To save parentheses, we assume that the modal operators bind strongest. For example, $\Diamond p \wedge q = (\Diamond p) \wedge q$. We only consider formulas in negation normal form. Formulas with general negation can be transformed to negation normal form in linear time using the propositional De Morgan's laws and the equations $\neg\Diamond s = \Box\neg s$ and $\neg\Box s = \Diamond\neg s$.

We give the relational semantics of K. The models of formulas are transition systems whose states are labeled with predicates. The modal formula $p$ is satisfied by a state in a model iff this state is labeled with $p$. The formula $\Diamond s$ is satisfied by a state iff there is a successor state that satisfies $s$. The formula $\Box s$ is satisfied by a state iff all successor states satisfy $s$.

Formally, a model $\mathcal{M}$ consists of the following components:

- A nonempty set $|\mathcal{M}|$ of **states**.

$$M_\wedge \frac{\sigma{:}s_1 \wedge \cdots \wedge s_n}{\sigma{:}s_i} \quad M_\vee \frac{\sigma{:}s \vee t}{\sigma{:}s \mid \sigma{:}t}\, i{\in}[1,n] \quad M_\otimes \frac{\sigma{:}s,\ \sigma{:}\neg s}{}$$

$$M_\Diamond \frac{\sigma{:}\Diamond s}{\sigma s{:}s} \quad M_\Box \frac{\sigma{:}\Box s,\ \sigma t{:}t}{\sigma t{:}s}$$

Figure 5.1: Tableau system for K

- A **transition relation** $\to_\mathcal{M} \subseteq |\mathcal{M}| \times |\mathcal{M}|$.
- For every predicate $p$, the set of states $\mathcal{M}p \subseteq |\mathcal{M}|$ that are **labeled** with $p$.

The relation $\mathcal{M}, v \models s$ is the set of all triples such that the state $v \in |\mathcal{M}|$ **satisfies** $s$. It is defined by induction on the structure of $s$.

$$\begin{aligned}
\mathcal{M}, v \models p \quad &:\Leftrightarrow \quad v \in \mathcal{M}p \\
\mathcal{M}, v \models \neg p \quad &:\Leftrightarrow \quad v \notin \mathcal{M}p \\
\mathcal{M}, v \models s \wedge t \quad &:\Leftrightarrow \quad \mathcal{M}, v \models s \text{ and } \mathcal{M}, v \models t \\
\mathcal{M}, v \models s \vee t \quad &:\Leftrightarrow \quad \mathcal{M}, v \models s \text{ or } \mathcal{M}, v \models t \\
\mathcal{M}, v \models \Diamond s \quad &:\Leftrightarrow \quad \text{there is some } v' \in |\mathcal{M}| \text{ such that } v \to_\mathcal{M} v' \text{ and } \mathcal{M}, v' \models s \\
\mathcal{M}, v \models \Box s \quad &:\Leftrightarrow \quad \text{for all } v' \in |\mathcal{M}| \text{ with } v \to_\mathcal{M} v' \text{ we have } \mathcal{M}, v' \models s
\end{aligned}$$

A formula $s$ is called **satisfiable** if there is a model $\mathcal{M}$ and a state $v \in |\mathcal{M}|$ such that $\mathcal{M}, v \models s$.

## 5.2 Rules

Now we will define a tableau system for K and show how it fits into our framework. We use **prefixed formulas** $\sigma{:}s$ that consist of a prefix $\sigma$ and a modal formula $s$. Every prefix $\sigma$ stands for a state. The prefixed formula $\sigma{:}s$ means that the state represented by $\sigma$ satisfies $s$. A **prefix** $\sigma$ is a possibly empty sequence of formulas $\sigma = s_1 s_2 \cdots s_n$. The empty prefix is written as $\varepsilon$.

The elements of $\mathcal{F}$, that is, the objects on the tableau, are prefixed formulas $\sigma{:}s$. To determine whether a modal formula $s$ is satisfiable, we build a tableau starting with the prefixed formula $\varepsilon{:}s$.

The rule schemas in Figure 5.1 describe a tableau system for K. The main difference to the tableau systems by Fitting [7] and Massacci [23] is that these systems

use sequences of numbers as prefixes and require prefixes introduced by the equivalent of the $M_\Diamond$-rule to be **fresh**, that is, not to appear on the branch so far. It is impossible to directly express freshness in our framework. However, our rules ensure the necessary freshness conditions by construction without the need for explicit global conditions. This is achieved by computing a new prefix from the $\Diamond$-formula that created it in a way that it cannot collide with existing prefixes. This idea also appears in [3]. See Section 6.3 for further discussion.

In the following, we argue the soundness, completeness and termination of the tableau system.

## Soundness

We have to show that the rules maintain the satisfiability of branches. From this, it follows that satisfiable branches are consistent. A branch $B$ is **satisfiable** if there is a model $\mathcal{M}$ and an interpretation $[\cdot]$ that maps every prefix $\sigma$ to a state of the model $[\sigma]$ such that $[\sigma]$ satisfies $s$ for all $\sigma{:}s \in B$ and $[\sigma] \rightarrow_{\mathcal{M}} [\sigma s]$ whenever $\sigma s{:}s \in B$. Thus the branch $\{\varepsilon{:}s\}$ is satisfiable iff $s$ is satisfiable.

We only show the soundness of $M_\Diamond$ and $M_\Box$. The soundness of the other rules can be verified easily.

First note that a prefix $\sigma s$ can only be introduced by an application of $M_\Diamond$ to the prefixed formula $\sigma{:}\Diamond s$ that adds $\sigma s{:}s$ to the branch. Thus if $M_\Diamond$ adds $\sigma s{:}s$ to a branch, then the prefix $\sigma s$ did not appear on the branch before (that is, there was no $u$ such that $\sigma s{:}u$ was on the branch).

The rule $M_\Diamond$ creates a new prefix for the successor state required by a $\Diamond$-formula. Consider a step $\frac{B}{\{\sigma s:s\}}$ where $\sigma{:}\Diamond s \in B$ and $B$ is satisfiable. We have to show that the branch $B \cup \{\sigma s{:}s\}$ is satisfiable. Since $\sigma{:}\Diamond s \in B$, we have that $[\sigma]$ satisfies $\Diamond s$. Thus there has to be a successor of $[\sigma]$ that satisfies $s$. Because $\frac{B}{\{\sigma s:s\}}$ is a step, we have $\sigma s{:}s \notin B$. Hence the prefix $\sigma s$ does not appear on $B$. Thus we can interpret $\sigma s$ as a successor of $[\sigma]$ where $s$ holds. Hence $B \cup \{\sigma s{:}s\}$ is satisfiable.

The rule $M_\Box$ propagates a $\Box$-formula to a prefix that stands for a successor state. Consider a step $\frac{B}{\{\sigma t:s\}}$ where $\sigma{:}\Box s \in B$, $\sigma t{:}t \in B$ and $B$ is satisfiable. Then $[\sigma]$ satisfies $\Box s$. Thus all successors of $[\sigma]$ satisfy $s$. Because $\sigma t{:}t \in B$, we have that $[\sigma] \rightarrow_{\mathcal{M}} [\sigma t]$. Thus $[\sigma t]$ satisfies $s$. Hence $B \cup \{\sigma t{:}s\}$ is satisfiable.

## Completeness

To prove completeness, one can show that every evident branch $B$ corresponds to a model $\mathcal{M}$ that satisfies every formula on the branch. From this, it follows that consistent branches are satisfiable. The model is constructed as follows. The set of states $|\mathcal{M}|$ is the set of prefixes that appear on the branch. A state $\sigma$ is labeled

with $p$, that is $\sigma \in \mathcal{M}p$, iff $\sigma{:}p \in B$. The transition relation is obtained from the structure of the prefixes. We set $\sigma \rightarrow_{\mathcal{M}} \sigma s$ iff $\sigma s{:}s \in B$. Then one can show that $\mathcal{M}, \sigma \models s$ for all $\sigma{:}s \in B$ by induction on the size of $s$.

### Termination

The tableau system terminates. First note that all modal formulas and all elements of prefixes are subformulas of the initial formula. Thus it suffices to show that the length of the prefixes is bounded. We write $|\sigma|$ for the length of a prefix $\sigma$, i.e., the number $n$ such that $\sigma = s_1 \cdots s_n$. We define the **modal depth** $d(s)$ of a formula $s$ as follows.

$$\begin{array}{rcl} d(p) & = & 0 \\ d(\neg p) & = & 0 \\ d(s \wedge t) & = & \max\{d(s), d(t)\} \\ d(s \vee t) & = & \max\{d(s), d(t)\} \\ d(\Diamond s) & = & 1 + d(s) \\ d(\Box s) & = & 1 + d(s) \end{array}$$

Let $d_0$ be the modal depth of the initial formula. We have the following invariant for all prefixed formulas $\sigma{:}s$ on the tableau.

$$|\sigma| + d(s) \leq d_0$$

Thus the length of all prefixes is bounded by $d_0$.

Since the tableau system terminates and a branch is consistent iff it is satisfiable, the tableau system fits in our framework.

## 5.3 Example

An example of a tableau for the formula $s := \Diamond((\Box q_1 \vee \Diamond \neg q_2) \wedge (p_1 \vee p_2) \wedge \Diamond q_2 \wedge \Box(\neg q_1 \vee \neg q_2))$ is depicted in Figure 5.5. Note that $sq_2 : q_2$ and $s\neg q_2 : \neg q_2$ do not clash because they have different prefixes. Branch $e$ is evident. It contains the prefixes $\varepsilon$, $s$, $sq_2$ and $s\neg q_2$. Thus this branch corresponds to a model with 4 states, which is depicted in Figure 5.3. Each state is annotated with the predicates that hold at the state.

| | | |
|---|---|---|
| 1 | $\varepsilon : s$ | |
| 2 | $s : (\Box q_1 \vee \Diamond\neg q_2) \wedge (p_1 \vee p_2) \wedge \Diamond q_2 \wedge \Box(\neg q_1 \vee \neg q_2)$ | $M_\Diamond(1)$ |
| 3 | $s : \Box q_1 \vee \Diamond\neg q_2$ | $M_\wedge(2)$ |
| 4 | $s : p_1 \vee p_2$ | $M_\wedge(2)$ |
| 5 | $s : \Diamond q_2$ | $M_\wedge(2)$ |
| 6 | $s : \Box(\neg q_1 \vee \neg q_2)$ | $M_\wedge(2)$ |
| 7 | $sq_2 : q_2$ | $M_\Diamond(5)$ |
| 8 | $sq_2 : \neg q_1 \vee \neg q_2$ | $M_\Box(6,7)$ |

Left branch:

| | | |
|---|---|---|
| 9 | $s : \Box q_1$ | $M_\vee(3)$ |
| 10 | $sq_2 : q_1$ | $M_\Box(9,7)$ |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | $s : p_1$ | $M_\vee(4)$ | | | | 16 | $s : p_2$ | $M_\vee(4)$ | | | |
| 12 | $sq_2 : \neg q_1$ | $M_\vee(8)$ | 14 | $sq_2 : \neg q_2$ | $M_\vee(8)$ | 17 | $sq_2 : \neg q_1$ | $M_\vee(8)$ | 19 | $sq_2 : \neg q_2$ | $M_\vee(8)$ |
| 13 | $\otimes$ | $M_\otimes(10,12)$ | 15 | $\otimes$ | $M_\otimes(7,14)$ | 18 | $\otimes$ | $M_\otimes(10,17)$ | 20 | $\otimes$ | $M_\otimes(7,19)$ |
| | $a$ | | | $b$ | | | $c$ | | | $d$ | |

Right branch:

| | | |
|---|---|---|
| 21 | $s : \Diamond\neg q_2$ | $M_\vee(3)$ |
| 22 | $s\neg q_2 : \neg q_2$ | $M_\Diamond(21)$ |

| | | | | |
|---|---|---|---|---|
| 23 | $s : p_1$ | $M_\vee(4)$ | | $s : p_2$ |
| 24 | $sq_2 : \neg q_1$ | $M_\vee(8)$ | $sq_2 : \neg q_2$ | |
| | $e$ | | | |

where $s := \Diamond((\Box q_1 \vee \Diamond\neg q_2) \wedge (p_1 \vee p_2) \wedge \Diamond q_2 \wedge \Box(\neg q_1 \vee \neg q_2))$

Figure 5.2: Example of a tableau for modal logic

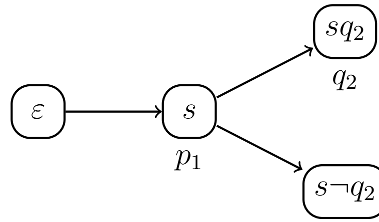


Figure 5.3: Model of the initial formula from Figure 5.2

Since the tableau system fits in our framework, we automatically obtain a backjumping procedure for it. Figure 5.4 shows the instantiation of the dependency sequent rules from Figure 4.3 for the modal tableau system. In Figure 5.5, we present an example of a tableau with backjumping for the same formula as in the tableau in Figure 5.2. Note that the backjump from branch $b$ to the branch $d$ results from a clash of formulas with prefix $sq_2$, but allows to jump over branching point 2, whose formulas have prefix $s$. Thus this backjumping calculus allows to jump over formulas with different prefixes.

$$D_0^{\text{mod}}\,\frac{}{\sigma : s \Rightarrow \sigma : s}$$

$$D_{\alpha,\wedge}^{\text{mod}}\,\frac{A \Rightarrow \sigma : s_1 \wedge \cdots \wedge s_n}{A \Rightarrow \sigma : s_i}\; {\scriptstyle i \in [1,n]}$$

$$D_{\alpha,\Diamond}^{\text{mod}}\,\frac{A \Rightarrow \sigma : \Diamond s}{A \Rightarrow \sigma s : s}$$

$$D_{\alpha,\Box}^{\text{mod}}\,\frac{A \Rightarrow \sigma : \Box s \quad A' \Rightarrow \sigma t : t}{A, A' \Rightarrow \sigma t : s}$$

$$D_{\beta}^{\text{prop}}\,\frac{A \Rightarrow \sigma : s \vee t \quad A', \sigma : s \Rightarrow \otimes}{A, A' \Rightarrow \sigma : t}$$

$$D_{\otimes}^{\text{prop}}\,\frac{A \Rightarrow \sigma : s \quad A' \Rightarrow \sigma : \neg s}{A, A' \Rightarrow \otimes}$$

Figure 5.4: Instantiation of the rules in Figure 4.3 for the modal tableau system

$$\varepsilon : s \Rightarrow \varepsilon : s$$
$$\varepsilon : s \Rightarrow s : (\Box q_1 \vee \Diamond \neg q_2) \wedge (p_1 \vee p_2) \wedge \Diamond q_2 \wedge \Box(\neg q_1 \vee \neg q_2)$$
$$\varepsilon : s \Rightarrow s : \Box q_1 \vee \Diamond \neg q_2$$
$$\varepsilon : s \Rightarrow s : p_1 \vee p_2$$
$$\varepsilon : s \Rightarrow s : \Diamond q_2$$
$$\varepsilon : s \Rightarrow s : \Box(\neg q_1 \vee \neg q_2)$$
$$\varepsilon : s \Rightarrow sq_2 : q_2$$
$$\varepsilon : s \Rightarrow sq_2 : \neg q_1 \vee \neg q_2$$

1 — branch (left):

$$s : \Box q_1 \Rightarrow s : \Box q_1$$
$$\varepsilon : s, s : \Box q_1 \Rightarrow sq_2 : q_1$$

2 — branch (left): $s : p_1 \Rightarrow s : p_1$ | $s : p_2$ (c)

3 — branch:

(a) $sq_2 : \neg q_1 \Rightarrow sq_2 : \neg q_1$; $\varepsilon : s, s : \Box q_1, sq_1 : \neg q_1 \Rightarrow \otimes$

(b) $\varepsilon : s, s : \Box q_1 \Rightarrow sq_2 : \neg q_2$; $\varepsilon : s, s : \Box q_1 \Rightarrow \otimes$

1 — branch (right):

$$\varepsilon : s \Rightarrow s : \Diamond \neg q_2$$
$$\varepsilon : s \Rightarrow s \neg q_2 : \neg q_2$$

4 — branch: $s : p_1 \Rightarrow s : p_1$ | $s : p_2$

5 — branch: $sq_2 : \neg q_1 \Rightarrow sq_2 : \neg q_1$ | $sq_2 : \neg q_2$

(d)

where $s := \Diamond((\Box q_1 \vee \Diamond \neg q_2) \wedge (p_1 \vee p_2) \wedge \Diamond q_2 \wedge \Box(\neg q_1 \vee \neg q_2))$

Figure 5.5: Tableau from Figure 5.2 extended with backjumping

# 6 Discussion and Future Work

We have formalized and proven correct backjumping with dependency sets for a class of tableau systems and introduced dependency sequents as a semantic characterization of dependency sets.

Now, we want to discuss possible extensions of our approach.

## 6.1 Blocking

Prefixed tableau systems for modal logics often employ a blocking technique to ensure termination (e.g., [5, 18]). Blocking restricts the applicability of certain tableau rules to a branch based on structural properties of the branch.

Interestingly, blocking does not interfere with backjumping. As long as blocking does not forbid all applicable rules, it acts like a rule selection strategy. Hence this does not affect the correctness of backjumping. When blocking does not allow to apply any rule to a branch, then the branch is known to be satisfiable by some external theorem provided that the blocking conditions are correct. At this point, the algorithm stops and we have that the initial formula is satisfiable by the external theorem regardless of what backjumping did before.

So we expect that our approach extends to tableau systems with blocking.

## 6.2 Non-terminating Tableau Systems

For undecidable logics like first-order logic, all complete tableau systems are non-terminating. Non-terminating tableau systems need additional care because they require to consider infinite evident sets and the rule application strategy has to be fair, that is, no rule is ignored infinitely often. We conjecture that it is possible to extend the present approach to non-terminating tableau systems.

## 6.3 Monotonicity

Another outstanding issue is the limited expressiveness of tableau rules. By defining tableau steps as the closure $\hat{\mathcal{T}}$ of the set $\mathcal{T}$ of tableau rules, we ensured that

if a rule applies to a branch $B$, then it applies to every larger branch $B' \supseteq B$ (up to some restrictions, see Proposition 2.3). Thus a tableau rule that applies to a branch $B$ cannot require that a formula, say $s$, is not on the branch because the rule automatically applies to $B \cup \{s\}$ (provided that $B \cup \{s\}$ is not clashed and contains no conclusion of the rule). We call tableau systems whose steps satisfy Proposition 2.3 **monotone**. Monotonicity is essential for our correctness proof.

But there are non-monotone tableau systems. A particularly simple case of non-monotone tableau systems are tableau systems with freshness conditions. For example, the prefixed tableau systems by Fitting [7] and Massacci [23] impose fresh prefixes. They require that a prefix introduced by the $\Diamond$-rule is not on the branch before. In Chapter 5, we showed how this requirement can be avoided by computing prefixes in a way such that they cannot collide. This way, we obtained a monotone tableau system.

But there are more severe cases where it is not obvious how to make a tableau system monotone. For example, Schneider's [28] tableau system for modal logic with transitive closure uses a loop-check technique that requires lower as well as upper bounds on the sets of formulas at the nodes of a loop in order to cause a clash. Since an upper bound means that certain formulas are not on the branch, Schneider's system cannot be expressed within our current framework. Extending the present approach to non-monotone tableau systems is future work.

# Bibliography

[1] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[2] J. Backes and C. E. Brown. Analytic tableaux for higher-order logic with choice. In *IJCAR 2010*, volume 6173 of *LNCS/LNAI*, pages 76–90. Springer, 2010.

[3] B. Beckert and R. Goré. Free variable tableaux for propositional modal logics. In *TABLEAUX 97*, volume 1227 of *LNCS/LNAI*, pages 91–106. Springer, 1997.

[4] E. W. Beth. Semantic entailment and formal definability. *Mededelingen der Koninklijke Nederlandse Akademie van Wetenschappen*, 18(13):309–342, 1955.

[5] T. Bolander and P. Blackburn. Termination for hybrid tableaus. *Journal of Logic and Computation*, 17(3):517–554, 2007.

[6] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors. *Handbook of tableau methods*. Kluwer Academic Publishers, 1999.

[7] M. Fitting. *Proof methods for modal and intuitionistic logics*. Springer, 1983.

[8] J. Gaschnig. Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisfying Assignment Problems. In *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*, pages 268–277, 1978.

[9] M. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[10] D. Götzmann, M. Kaminski, and G. Smolka. Spartacus: A tableau prover for hybrid logic. *Electronic Notes in Theoretical Computer Science*, 262:127–139, 2010.

[11] V. Haarslev and R. Müller. Racer system description. In *IJCAR 2001*, volume 2083 of *LNCS/LNAI*, pages 701–705. Springer, 2001.

[12] K. Hintikka. Form and content in quantification theory. *Acta Philosophica Fennica*, 8:7–55, 1955.

[13] G. Hoffmann and C. Areces. HTab: a terminating tableaux system for hybrid logic. *Electronic Notes in Theoretical Computer Science*, 231:3–19, 2009.

[14] I. Horrocks. *Optimising tableaux decision procedures for description logics.* PhD thesis, University of Manchester, 1997.

[15] I. Horrocks. Implementation and optimization techniques. In *The description logic handbook*, pages 306–346. Cambridge University Press, 2003.

[16] I. Horrocks and P. Patel-Schneider. FaCT and DLP. In *TABLEAUX 98*, volume 1397 of *LNCS/LNAI*, pages 27–30. Springer, 1998.

[17] I. Horrocks and P. F. Patel-Schneider. Optimizing description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

[18] I. Horrocks, U. Hustadt, U. Sattler, and R. Schmidt. Computational modal logic. In *Handbook of Modal Logic*. Elsevier, 2006.

[19] U. Hustadt and R. Schmidt. Simplification and backjumping in modal tableau. In *TABLEAUX 98*, LNCS/LNAI, pages 187–201. Springer, 1998.

[20] M. Kaminski and G. Smolka. Terminating tableaux for hybrid logic with eventualities. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS/LNAI*, pages 240–254. Springer, Jul 2010.

[21] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM J. Comput.*, 6(3):467–480, 1977.

[22] Z. Lis. Wynikanie semantyczne a wynikanie formalne. *Studia Logica*, 10: 39–60, 1960.

[23] F. Massacci. Single step tableaux for modal logics. *Journal of Automated Reasoning*, 24(3):319–364, 2000.

[24] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85(112-133):12, 2005.

[25] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM*, 53(6):937–977, 2006.

[26] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[27] F. Rossi, P. Van Beek, and T. Walsh, editors. *Handbook of constraint programming*. Elsevier, 2006.

[28] S. Schneider. *Terminating Tableaux for Modal Logic with Transitive Closure*. Bachelor's thesis, Saarland University, 2009.

[29] R. Smullyan. *First-Order Logic*, volume 43 of Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer, 1968.

[30] R. Stallman and G. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[31] D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *IJCAR 2006*, volume 4130 of *LNCS/LNAI*, pages 292–297. Springer, 2006.