Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

# Bachelor Thesis Informatics

## Logik-Lehrtools

Jose Carlos Espinoza Quiroz

31. July 2018

**Reviewer**

Prof. Dr. sc. techn. (ETH) Wolfgang Küchlin
Department of Computer Science
University of Tübingen

**Espinoza Quiroz, Jose Carlos:**
*Logik-Lehrtools*
Bachelor Thesis Informatics
Eberhard Karls Universität Tübingen
Period: 1. April 2018 - 31 July 2018

# Abstract

In this thesis we will discuss the implementation of a JavaFX application allowing a step by step demonstration of several algorithms used to test the statisfiability of propositional formulas. The intended use of this application is as material to accompany computer science lectures discussing these algorithms. The algorithms that have been implemented are the Resolution algorithm, Back Dual Resolution algorithm and Davis-Putnam algorithm from 1960.

# Acknowledgements

I take this opportunity to express gratitude to Herr Prof. Dr. Küchlin from the Department of Computer Science at the University of Tübingen for giving me a chance to write a thesis and for his help and support.

I wish to express my sincere thanks to Frau Kahl for providing a second home in Metzingen for me during the time away from my family.

I also thank my parents for the constant encouragement, support and attention.

Last but not least I am also grateful to my wife and my daughter who continuously helped and supported me throughout my bachelor studies.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

**CNF** Conjunctive Normal Form
**CSS** JavaFX Cascading Style Sheets
**DNF** Disjunctive Normal Form
**DP60** the first Davis-Putnam Algorithm from 1960
**GUI** Graphical User Interface
**MVC** Model-View-Controler
**MVP** Model-View-Presenter
**MVVM** Model-View-View model
**UI** User Interface

# Chapter 1

# Introduction

## 1.1 Motivation

The idea behind this project was the need to have a tool accompanying the Logic lectures offered by the Department of Symbolic Computation at the Wilhelm-Schickard-Institut für Informatik of the Eberhard Karls University of Tübingen.
The idea was develop an application that could be used during the lecture to demonstrate live how certain algorithm works. The goal was to implement the functionality to demonstrate the algorithm that tests the satisfiability of a propositional formula, mainly the Resolution Algorithm, Back Dual Algorithm, the first Davis-Putnam Algorithm from 1960 and the Davis-Putnam-Logemann-Loveland Algorithm.
Also this application will help on writing examples for the different lectures concerned with this algorithm, since doing this by hand is a tedious and time consuming work.

We are going to start with an introduction to the theory and structures used across the whole document in chapter 2. In Chapter 3 the implementation of the Main Menu of the application is discussed. The Chapters 4 ,5 and 6 contain the description for the Resolution algorithm , Back Dual Resolution algorithm and the DP60 algorithm. The last chapter, Chapter 7 will contain the conclusions and outlook.

## 1.2 Material

This application was implemented as a JavaFX application. JavaFX is an extension of Java used to create rich graphical user interfaces with ease.
Initially the integrated development environment for writing the application

was Eclipse, but after running into several problem with the integration of JavaFX it was decided to switch to IntelliJ IDEA.
The Java version 9 was used to program the application. Oracle offers a visual layout tool named JavaFX SceneBuiler2.0 letting the user design JavaFX user interfaces with ease, but since the 2.0 version was designed for Java 8 it did not work optimaly with Java 9. An alternative used instead of the Scene Builder from Oracle was the SceneBuilder from Gluon, which is basically an updated version of Oracles SceneBuilder, which has more compatibility with Java 9.

# Chapter 2

# Theory and Structures

## 2.1 Theory

### 2.1.1 Propositional Logic

At the base of the propositional logic are the **atomic propositions**, i.e. expressions that can not be simplified and are used to build more complex expressions or formulas. Atomic propositions can be true or false and are represented as *propositional* variables. The values **true** ($\bot$) and **false** ($\top$) are constants and thus considered atomic expressions. In order to build up more complex expressions the logical operators are used as follows:

**Definition 1.** Any atomic expression is a propositional formula.
If $A$ and $B$ are propositional formulas, then $A \odot B$, where

$$\odot \in \{\wedge,\ \vee,\ \rightarrow,\ \leftrightarrow,\ \oplus\}$$

is also a propositional formula.
If $A$ is a propositional formula, then $\neg A$ is also a propositional formula.
If $A$ is a propositional formula, then $(A)$ is also a propositional formula.

For the rest of this section propositional variables will be written in lower case and propositional formulas in upper case.

In order to evaluate a propositional formula, truth values need to be assigned to all the propositional variables. The truth values are a pair of values allowing the formula to be evaluated, the most commonly used pairs are {true, false} , $\{t, f\}$ or $\{0, 1\}$.

Evaluating a propositional formula is formally known as the **interpretation of a propositional formula**.

**Definition 2.** Let $F$ be a propositional formula, $Var(F)$ be the set of variables appearing in $F$ and $B$ the set of truth values.
Then a **truth assignment** or **interpretation** for $F$ is a total function:

$$\beta_0 : Var(F) \to B.$$

Given a set of variables $Var(F) = \{a_1,\ a_2,\ \ldots,\ a_n\}$ from a proposition formula $F$, then a **truth assignment** is represented as an ordered $n$-tuple

$$(\beta_0(a_1),\ \beta_0(a_2),\ \ldots,\ \beta_0(a_n)).$$

Based on this representation, then a truth assignment is an element of the set $B^{|Var(F)|}$.

Furthermore

**Definition 3.** Let $B$ be the set of truth values and $F$ a propositional formula. Then the **truth value** of $F$ is a function

$$v(F) : B^{|Var(F)|} \to B.$$

For a specific truth assignment $\beta_0$ the truth value will be denoted by $v_{\beta_0}(F)$.

The function $v$ is also called the **valuation function** of $F$.

Truth tables are a convenient way of showing the truth values of every possible truth assignment, see Table 2.1 .

| $P$ | $Q$ | $\neg P$ | $P \wedge Q$ | $P \vee Q$ | $P \to Q$ | $P \leftrightarrow Q$ | $P \oplus Q$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

**Table 2.1:** Truth table for the operators in Definition 1

**Definition 4.** Let $B$ be the set of truth values, $\Omega = \{\neg, \wedge, \vee, \to, \leftrightarrow, \oplus\}$ be the set of operators and $\Gamma = \{v | v : B \times B \to B\} \cup \{v | v : B \to B\}$.
Then the function $\sigma : \Omega \to \Gamma$ is a mapping of each operator to its respective valuation function.

**Definition 5.** Let $F$ be a propositional formula, $\beta_0$ a truth assignment and $B = \{0, 1\}$ the set of truth values.

The truth value of $F$ for a truth assignment $\beta_0$ is defined inductively as
$v_{\beta_0}(\text{true}) = 1,\ v_{\beta_0}(\text{false}) = 0$
$v_{\beta_0}(F) = \beta_0(F)$ if $F$ is a variable.
$v_{\beta_0}(F) = \sigma(\odot)(v_{\beta_0}(A))$ if $F = \neg A$.
$v_{\beta_0}(F) = \sigma(\odot)(v_{\beta_0}(A), v_{\beta_0}(B))$ if $F = A \odot B$, where $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$.

**Definition 6.** A propositional formula $F$ is called **satisfiable** if there is a truth assignment $\beta_0$ such that $v_{\beta_0}(F) = 1$.
A truth assignment for which a propositional formula is satisfiable is called a **model** of $F$ and denoted by $\beta_0 \models F$.

If for a propositional formula there exists no truth assignment such that $v_{\beta_0}(F) = 1$ then that formula is called **unsatisfiable**.

**Definition 7.** A propositional formula is called **valid** or **tautology** if for all possible truth assignments $\beta_0$ the truth value $v_{\beta_0}(F)$ is always 1 and denoted by $\models F$.

The definition for a set of formulas are analogous to the ones for a single formula as stated in the next definitions.

**Definition 8.** Let $U = \{A_1, A_2, \ldots\}$ a set of propositional formula, $B$ be the set of truth values and $Var(U) = \cup_{i=1,2,\ldots} A_i$ be the set of all variables in $U$.
Then a truth assignment is defined as a total function $\beta_0 : Var(U) \rightarrow B$.

The truth value for a propositional formula $A_i \in U$ is defined as for the single formula case.

**Definition 9.** Let $U = A_1, A_2, \ldots$ be a set of propositional formulas and $B$ be the set of truth values.
The set of formulas $U$ is called **satisfiable** if there exists a truth assignment $\beta_0$ for $U$ such that for all $A_i \in U$, $v_{\beta_0}(A_i) = 1$.
$\beta_0$ is called **a model**.
If $U$ is not satisfiable then its called **unsatisfiable**.

**Definition 10.** Let $A_1$ and $A_2$ be propositional formulas, then $A_1$ is equivalent to $A_2$, denoted by $A_1 \equiv A_2$, if for every truth assignment $\beta_0$, $v_{\beta_0}(A_1) = v_{\beta_0}(A_2)$.

**Theorem 1.** *$A1 \equiv A2$ if and only if $A1 \leftrightarrow A2$ is true in every interpretation.*

A proof to this theorem can be found in [4] on Page 22.

An interesting consequence of this theorem is that it is possible to define one logical operator based on other operators, e.g.

$$\begin{aligned} A \rightarrow B &\equiv \neg A \vee B \\ A \leftrightarrow B &\equiv (A \rightarrow B) \wedge (B \rightarrow A) \equiv (\neg A \vee B) \wedge (\neg B \vee A) \\ A \oplus B &\equiv \neg(A \rightarrow B) \vee \neg(B \rightarrow A) \equiv \neg(\neg A \vee B) \wedge \neg(\neg B \vee A) \end{aligned}$$

These equivalences can be easily verified, thus it is possible to write every propositional formula in function of only three logical operators: negation, disjunction and conjunction. From now on we will only discuss properties of propositional formulas formed with these 3 operators.

Some other important equivalences are listed below:

Commutativity: $A \vee B \equiv B \vee A$ and $A \wedge B \equiv B \wedge A$

Associativity: $A \vee (B \vee C) \equiv (A \vee B) \vee C$ and $A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$

Distributivity:
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ and $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

De Morgan law: $A \vee B \equiv \neg(\neg A \wedge \neg B)$ and $A \wedge B \equiv \neg(\neg A \vee \neg B)$

Absorption: $F \vee (F \wedge G) \equiv F \wedge (F \vee G) \equiv F$

Absorption of constants:
$A \vee$ `true` $\equiv$ `true`; $A \wedge$ `false` $\equiv A$; $A \wedge$ `true` $\equiv A$; $A \wedge$ `false` $\equiv$ `false`

**Definition 11.** Let $U = A_1, A_2, \ldots$ a set of propositional formulas and $A$ is a propositional formula.
A is called **a logical consequence** of $U$, denoted by $U \models A$, if every model from $U$ is also a model from $A$.

A consequence of Definition 11 is the following statement:

$$U \cup \{F\} \models G \text{ if and only if } U \models F \rightarrow G.$$

The first step towards a normal form is to eliminate all constants from the propositional formula. This can be achieved by successively applying the absorption of constants as well as following equivalences for the negation:

$\neg$ `true` $\equiv$ `false`

$\neg$ `false` $\equiv$ `true`

$(\neg F) \equiv F$, for any propositional formula $F$.

This form is denoted as **the simplified form of a propositional formula** and is equivalent to the original propositional formula.

**Definition 12.** A **literal** is an atomic expression or a negation of one. A Formula $F$ is in **negation normal form (NNF)** if it is in a simplified form and negation only appears as a literal, i.e., negation is only applied to atomic expressions.

A literal composed only by an atomic expression is called **a positive literal** and the negation of an atomic expression **a negative literal**.
A positive literal and a negative literal of the same atomic expression are called **complementary literals**. The NNF is achieved by calculating the simplified form of the formula and then successively applying the de Morgan law and the simplifying rule, e.g. $(\neg F) \equiv F$.

**Definition 13. A clause** is a formula of the form $\bigvee_{i=1}^{n} A_i$, where $A_i$ is a literal for all $i = 1 \dots n$. **A term** is a formula of the for $\bigwedge_{j=1}^{m} A_j$, where $A_j$ is a literal for all $j = 1 \dots n$.

**Definition 14.** A formula $F$ is said to be in **conjunctive normal form (CNF)** if it is a conjunction of clauses

$$\bigvee_{i=1}^{n} A_i,$$

where $A_i$ is a clause for $i = 1 \dots n$.

In order to create an equivalent propositional formula in CNF, we need to first create an equivalent formula in NNF, then apply the distributivity laws as follows:

Substitute $A \vee (B \wedge C)$ with $(A \vee B) \wedge (A \vee C)$

Substitute $(B \wedge C) \vee A$ with $(B \vee A) \wedge (C \vee A)$,

until an equivalent formula in CNF is obtained. Both rules are equivalent, the only difference is a consequence of applying the commutative law.

**Definition 15.** A formula $F$ is said to be in **disjunctive normal form (DNF)** if it's a disjunction of terms,

$$\bigwedge_{i=1}^{n} A_i,$$

where $A_i$ is a term for $i = 1 \dots n$.

In order to create an equivalent propositional formula in DNF, we need to first create an equivalent formula in NNF, then apply the distributivity laws as follows:

Substitute $A \wedge (B \vee C)$ with $(A \wedge B) \vee (A \wedge C)$

Substitute $(B \vee C) \wedge A$ with $(B \wedge A) \vee (C \wedge A)$,

until an equivalent formula in DFN is obtained. Both rules are equivalent, the only difference is a consequence of applying the commutative law.

For propositional formulas in CNF or DNF there is a simplified notation, the set-based representation. Here a clause or a term is represented as a set of the literals it contains and a propositional formula in CNF is then a set of clauses and in DNF is a set of terms, where the clauses and terms are represented as sets as well. Written in a more formal was: A formula in CNF has the following form

$$\begin{aligned}(a_{i_1} \vee a_{i_2} \vee \ldots \vee a_{i_l}) \wedge (a_{j_1} \vee a_{j_2} \vee \ldots \vee a_{j_n}) \wedge \ldots \wedge (a_{k_1} \vee a_{k_2} \vee \ldots \vee a_{k_n}) \\ = \{\{a_{i_1}, a_{i_2}, \ldots, a_{i_l}\}, \{a_{j_1}, a_{j_2}, \ldots, a_{j_m}\}, \ldots, \{a_{k_1}, a_{k_2}, \ldots, a_{k_m}\}\} \quad (2.1)\end{aligned}$$

The DNF representation of a formula is as follows

$$\begin{aligned}(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_l}) \vee (a_{j_1} \wedge a_{j_2} \wedge \ldots \wedge a_{j_m}) \vee \ldots \vee (a_{k_1} \wedge a_{k_2} \wedge \ldots \wedge a_{k_n}) \\ = \{\{a_{i_1}, a_{i_2}, \ldots, a_{i_l}\}, \{a_{j_1}, a_{j_2}, \ldots, a_{j_l}\}, \ldots, \{a_{k_1}, a_{k_2}, \ldots, a_{k_l}\}\} \quad (2.2)\end{aligned}$$

This way of representing a propositional formula in CNF or DNF is used in this application.

Two clauses $C_1$ and $C_2$ are called **clashing clauses** if for two complementary literals $L_1$ and $L_2$,

$$L_1 \in C_1 \text{ and } L_2 \in C_2.$$

From now on, we assume that all formulas in CNF or DNF are in a set based representation.

A clause that does not contain any literal is called **the empty clause** and is often represented by $\square$. The empty clause is unsatisfiable.
On the other hand, the formula represented by an empty set of clauses is a tautology. A clause containing complementary literals is called **a trivial clause** and is also a tautology.

### 2.1.2 Resolution

**Definition 16.** Let $C_1$, $C_2$ and $R$ be clauses, where $C_1$ and $C_2$ are clashing clauses with complementary literals $L_1$ and $L_2$,
then $R$ is called **the resolvent** of $C_1$ and $C_2$ if $R = \{C_1 - \{L_1\}\} \cup \{C_2 - \{L_2\}\}$.
$C_1$ and $C_2$ are called **the parent clauses** of $R$.

Its relevant to mention that the resolvent $R$ is satisfiable if and only if the parent clauses are both satisfiable too.

A proof to this statement is found in [4] on page 81.
Furthermore if the empty clause is resolved then the parent clauses are unsatisfiable.

**Definition 17.** Let $F$ be a formula in CNF. We define inductively

$$Res(F)^0 = F$$

and the set of resolvents after deriving all possible resolvents $n$ times over $F$ as

$$Res(F)^n = Res(F)^{n-1} \cup \{R\}$$

where $R$ is not a trivial clause and is a resolvent of clashing clauses $C_1$, $C_2 \in Res(F)^{n-1}$

For a given propositional formula if $\square \in Res(F)^i$ for an $i \in \mathbb{N}$ then $F$ is unsatisfiable.

**Definition 18.** Let $C_1$ and $C_2$ be clauses such that $C_1 \subseteq C_2$, then $C_1$ **subsumes** $C_2$.

The property of subsumed clauses is that $C_1 \models C_2$, since every model for $C_1$ is a model for $C_2$.
**Resolution Algorithm:** $F$ is a propositional formula in CNF.
$i = 0$
$Res(F)^0 = F$
`repeat`

> **Resolution Step:** `For` $Res(F)^i$ derive all non trivial resolvents which have not been derived yet and store in $R^i$
>
> `If` $R^i$ contains the empty clause `then` $F$ is unsatisfiable and `end`
>
> **Forward Subsumption Step:** Remove all clauses from $R^i$ that are subsumed by a clause in $Res(F)^i$
>
> `If` $R^i$ is empty `then` $F$ is satisfiable and `end`
>
> **Backward Subsumption Step:** Remove all clauses from $Res(F)^i$ that are subsumed by a clause in $R^i$
>
> $i = i + 1$
>
> $Res(F)^i = Res(F)^{i-1} \cup R^{i-1}$

If a formula in set based representation is unsatisfiable, then the empty clause would be derived by the algorithm.

### Backward Dual Resolution

Similar as with the resolution, resolvents can be formed for terms and a resolution can be defined over a propositional formula in DNF and it is denoted as **the backward dual resolution**.

We will start with two terms $T_1$ and $T_2$ and a pair of complementary literals $L_1$ and $L_2$ such that $L_1 \in T_1$ and $L_2 \in T_2$, then the clause $R = T_1 - \{L_1\} \cup T_2 - \{L_2\}$ is called **resolvent**. The set of resolvents is defined exactly as for the resolution see Definition 17 , where $R$ is not a trivial term and resolvent of term $C_1, C_2 \in Res(F)^{n-1}$.
The trivial term here refers to the term containing complementary literals and its always false.

Just as important in this chapter is the absorption law applied to terms. For two terms $T_1$ and $T_2$ such that $T_1 \subseteq T_2$ follows that $T_1 \wedge T_2 \equiv T_1$, denoted as $T_1$ **absorbed** $T_2$.
So for a propositional formula in DNF containing two terms $T_1$ and $T_2$ such that $T_1 \subseteq T_2$, then removing $T_2$ will create an equivalent formula.
Formally if $F$ is a propositional formula in DFN and $T_1$, $T_2$ are terms of $F$ such that $T_1 \subseteq T_2$, then $F \equiv F - T_2$.

**Definition 19.** Let $F$ be a propositional formula in DNF and let $T$ be a term. $T$ is called an **implicant** of $F$ if $T \models F$.

**Definition 20.** Let $F$ be a propositional formula in DNF and let $T$ be an implicant of $F$. $T$ is called **a prime implicant** if the removal of any literal of $T$ results in it not being an implicant anymore.

Based on Definition 20 follows that an implicant $T$ is either a prime implicant or a subset of $T$ is a prime implicant.
**Backward Dual Algorithm:** $F$ is a propositional formula in DNF.
$i = 0$
$Res(F)^0 = F$ `repeat`

> `For` $Res(F)^i$ derive all non trivial resolvents that have not been derived yet and store in $R^i$
>
> Remove all terms from $R$ that are absorbed by a term in $Res(F)^i$
>
> `If` $R^i$ is empty `then end`
>
> Remove all terms from $Res(F)^i$ that are absorbed by a term in $R^i$
>
> $i = i + 1$

$$Res(F)^i = Res(F)^{i-1} \cup R^{i-1}$$

Letting this algorithm end for $i = n$, then $Res(F)^n$ will contain all prime implicants of $F$.
The disjunction of all the prime implicants is also called **disjunctive prime form**.
Most of the time not all prime implicants of $F$ are necessary to form an equivalent propositional formula, a subset of the prime implicants is often enough. This subset is called **the core prime implicants** and forms the minimal equivalent propositional formula of $F$ in DNF.

### The Davis Putnam Algorithm

This algorithm was one of the first algorithms to the decide satisfiability and it is the basis to one of the most successful sat solving algorithms.
The algorithm consists of three steps or rules, the one literal rule, the **pure literal** rule and the **variable elimination** rule. But before explaining the rules the following concepts are needed.

A unit clause is a clause that consists of one literal. In a propositional formula $F$ in CNF, a literal $L$ is called **a pure literal** if $L$ appears at least in one clause of $F$ but its complementary literal does not appear in any clause of $F$.

**Davis Putnam Algorithm:** Let $F$ be a propositional formula in CNF.
Apply the following rules repeatedly until its not possible anymore

> **Rule 1:** Unit Literal rule, `if` $F$ contains a unit clause $\{L\}$, `then` remove all clauses from $F$ that contain $L$ and remove the complementary literal of $L$ from all the clauses occuring in $F$.
>
> **Rule 2:** Pure Literal rule, `if` $F$ contains a pure literal, `then` remove all clauses containing the pure literal.

If Rules 1 and 2 can not be applied anymore, apply the following rule to $F$

> **Rule 3:** Variable Elimination rule, `for` a variable $a$ find all clashing clauses, i.e. all clauses containing either $a$ or the complementary literal of $a$. Derive all possible resolvents over these clauses, then remove the parent clauses from $F$ and add the resolvents to $F$.

Terminate if either the empty clause is derived or $F$ is the empty.
If the empty clause is derived then $F$ is unsatisfiable and if the $F$ is equivalent

to the empty set then it is satisfiable.

All the definitions, theorems and algorithms explained in this section were based on the book [4].

## 2.2 Structures

### 2.2.1 The Formula Class

The formula class is not just an ordinary class, it is a set of classes used to represent a propositional formula as a binary tree, where the inner nodes represent the operators and the leaves represent the variables. Since any propositional formula can be transformed into an equivalent propositional formula using only conjunction, disjunction and negation, the propositional formula is represented only with these operators. Implementation of these classes was part of the lecture "Praktikum Automatische Beweise: Grundlagen" and was mostly left unchanged.
Other than the constructors, the only functionality used was the conversion of the represented formula into a normal form. Two more methods where added to the original formula class, cnf_set_base_repesentation() and dnf_set_based_representation(), which convert the formula class into a Formula_NF class.

### 2.2.2 The Set Based Representation

This structure implements a set based representation of a propositional formula in CNF or DNF form (see Section2.1.1).
A set based representation contains a set of **clauses**, where the clauses contain a set of **literals**. A literal is a **variable** with a phase, to represent positive or negative variable.
At the base of the construct is the variable class corresponding to a variable in a propositional formula. The literal class is defined similarly with a variable at its core and a boolean variable to represent the phase of the literal. Furthermore the clause class is created by grouping literal instances into a set, then to a clause or a term of a propositional formula in normal form.
Until now these implementations were taken are based on the lecture "Praktikum Automatische Beweise: Grundlagen". At the top is the Formula_NF class that contains a set of Clauses. A copy constructor was added to all classes in order to create a deep copy, i.e. to take an instance of the same class as a parameter and generate a new class with the same content. This option was preferable compared to the clone-able interface due to the poor implementation offered by the Java programming language.

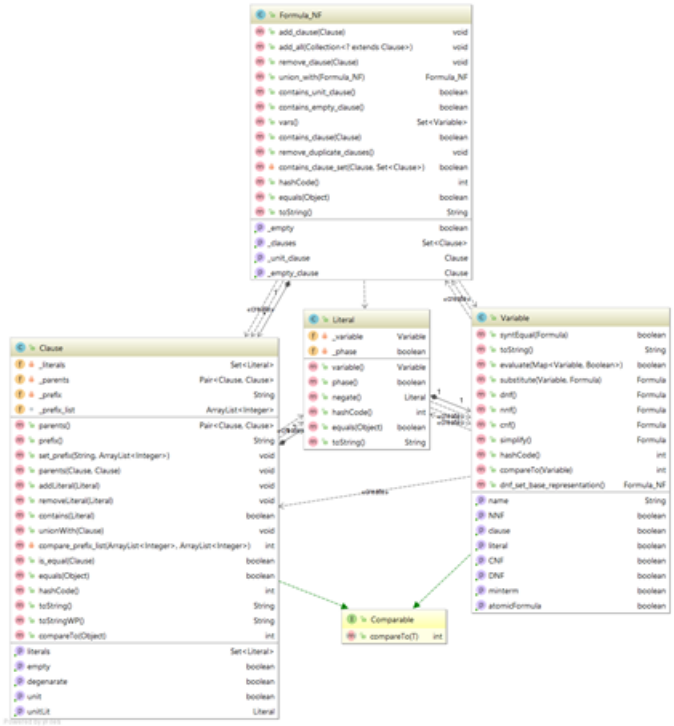

**Figure 2.1:** The Formula Class hierarchy

### The Clause Class

A set of literals was referred to the clause class, but the implementation needed much more functionality for the application. A Clause class contains a field Pair<Clause,Clause> _parents.
A string field named prefix containing a representation of _parents is also added to the Clause class.
The superseded `toString()` method will return a string representation of the clause with its corresponding prefix, but that is not always desired. Thus a second method `toStringWP()` was implemented, returning a string based representation of the clause without the prefix. This is necessary, e.g. when the prefix and the clause are to be displayed in two different colors.

### The Formula_NF class

The information is stored as a set of Clause instances. Like the clause, literal and variable classes, this class has a copy constructor also. It provides two methods to add clauses to the formula. One method adds one clause at a time and the other is adds a collection of clauses simultaneously.
This class also offers several methods to test if a clause or a specific clause is contained in the formula. These special clauses are the empty clause (see

Section 2.1.1) and the unit clause. It allows also to remove a specific clause, by using the `equals(Object)` method.
On the other hand the `remove_duplicates()` method searches for clause instances representing the same clause and stores the first appearance of each clause. Additional functionalities to merge formulas and to test empty formulas are provided too.
A last method offered by Formula_NF is the `vars()` method which returns a set of the variables contained in the formula.

### 2.2.3 Input Language and Parser

The syntax of the input strings representing a propositional formula is described by a formal Language. This section will introduce some basic concepts of the formal language theory for a better understanding on how the **Parser** is going to work.

Let $\Sigma$ be a finite nonempty set. In the context of words, $\Sigma$ is usually called an alphabet. The elements of $\Sigma$ are called symbols or letters.

A word $w$ over $\Sigma$ is a finite sequence of elements from $\Sigma$:

$$w : \{1, \ldots, l'\} \to \Sigma \quad \text{for some } l' \in N, \tag{2.3}$$

where $l'$ is called the length of $w$.
There is one word of length 0, the empty word, usually denoted by $\varepsilon$. Formally it is the sequence $\varnothing \to \Sigma$.
For example let $u : \{1, 2, 3\} \to \{a, b, c\}$ be defined by $u(1) = a$, $u(2) = b$ and $u(3) = a$. Then $u$ is a word over the alphabet $\{a, b, c\}$ of length 3.

The set of all words of length $n$ over $\Sigma$ is denoted by $\Sigma^n$. The set $\cup_{n=0}^{\infty}\Sigma^n$ of all finite words is denoted by $\Sigma^*$. Every formal language is a subset of $\Sigma^*$.

A grammar $G$ is described by a 4-tuple $(V, \Sigma, P, S)$, where

$V$ is a finite set, the set of variables or non-terminals.

$\Sigma$ is a finite set, the set of terminal symbols or alphabet, thus $V \cap \Sigma = \varnothing$.

$P$ is a finite subset of $(V \cup \Sigma)^* \times (V \cup \Sigma)^*$, the set of production rules.

$S \in V$ is the start variable.

If $(u, v) \in P$ is a production, then $u \to v$ will be used instead of $(u, v)$.

A formal language $L(G) \in \Sigma^*$ is the set of all words that can be formed by the grammar $G$.

Formal languages are categorized into 4 classes in a hierarchical manner as follows:
Let $G = (V, \Sigma, P, S)$ be a grammar.

1. Every grammar $G$ is a type-0 grammar.
2. $G$ is a **type-1 grammar** if $|u| \leq |v|$ for every production $u \to v \in P$. The only exception is the production $S \to \varepsilon$.
   If $S \to \varepsilon \in P$ then $S$ does not appear on the right-hand side of any production of $P$.
3. $G$ is a **type-2 grammar** if it is type-1 and the left-hand side of every production is an element from $V$, i.e. it is a non-terminal symbol.
4. $G$ is a **type-3 grammar** if it is type-2 and the right-hand side of every production is of the form $\Sigma V \cup \Sigma$.

A language $L \subseteq \Sigma^*$ is of type-$i$, $i = \{0, 1, 2, 3\}$ if there is grammar $G$ of type-$i$ such that $L(G) = L$.
The type-2 language is known as **context-free language** and respectively and the type-2 grammars are **context-free grammars**.

A **parse tree** is an entity which represents the structure of the derivation of a terminal string from the start variable $S$, i.e. a parse tree for a grammar $G$ is a tree where

- the root is the start symbol for $S$
- the interior nodes are the non-terminals symbols
- the leaf nodes are the terminal symbols
- the children of a node $T$ (from left to right) correspond to the symbols on the right hand side of some production for $T$.

Every terminal string generated by a grammar has a corresponding parse tree and every valid parse tree represents a string generated by the grammar. An example is provided below.
A grammar is $LL(1)$ if for each terminal $\alpha$ and non terminal $A$, there is some production $A \to \alpha$ with the following properties:
If $b_1 \dots b_n A\gamma$ appears in the left most derivation of some string $b_1 \dots b_n \alpha c_1 \dots c_m$ $(n, m \geq 0)$, the next derivation is necessarily $b_1 \dots b_n \alpha\gamma$.
A **predictive parser** is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. In order to accomplish this task, the predictive parser uses a look ahead pointer, which points to the next input symbols. This type of parser only accepts $LL(k)$ grammars.

### The Input Languages

The first grammar defined is the one representing a propositional formula. This representation requires 3 operators for `and`, `or` and `not`. It should also allow the use of braces.
The grammar was defined as follows $G = (V, \Sigma, P, S)$ where

$$
\begin{aligned}
V &= \{\texttt{EXPRESSION}, \texttt{TERM}, \texttt{OR\_OP}, \texttt{AND\_OP}, \texttt{FACTOR}, \texttt{ARGUMENT}\} \\
\Sigma &= \{\texttt{variable}, \texttt{or}, \texttt{and}, \texttt{not}, \texttt{open\_brace}, \texttt{close\_brace}\} \\
P &= \{ \\
&\quad \texttt{EXPRESSION} \rightarrow \texttt{TERM OR\_OP} \\
&\quad \texttt{OR\_OP} \rightarrow \texttt{or TERM OR\_OP} \mid \varepsilon \\
&\quad \texttt{TERM} \rightarrow \texttt{FACTOR AND\_OP} \\
&\quad \texttt{AND\_OP} \rightarrow \texttt{and FACTOR AND\_OP} \mid \varepsilon \\
&\quad \texttt{FACTOR} \rightarrow \texttt{not ARGUMENT} \mid \texttt{ARGUMENT} \\
&\quad \texttt{ARGUMENT} \rightarrow \texttt{variable} \mid \texttt{open\_brace EXPRESSION close\_brace}\}
\end{aligned}
\tag{2.4}
$$

In order to verify that this grammar actually corresponds to an $LL(1)$ grammar, the online tool on [1] was used.
The terminal symbol variable actually represents a string with the variable name. In the next section about lexical analysis this string will be treated as one entity. The terminal symbols `and`, `or` and `not` have not been defined yet, a specific value for these operators will be defined by each input language described in section 2.1.1.
The second grammar defined is the one used for the set based representation of a propositional formula in normal form.
The grammar is defined as $G' = (V', \Sigma', P', S')$ where

$$
\begin{aligned}
V' &= \{\texttt{NORMALFORM}, \texttt{CLAUSES}, \texttt{CLAUSE}, \texttt{LITERALS}, \texttt{LITERAL}\} \\
\Sigma &= \{\texttt{variable}, \texttt{not}, \texttt{comma}, \texttt{open\_curly\_bracket}, \texttt{close\_curly\_bracket}\} \\
P &= \{ \\
&\quad \texttt{NORMALFORM} \rightarrow \texttt{open\_curly\_bracket CLAUSE CLAUSES} \\
&\quad \texttt{CLAUSES} \rightarrow \texttt{comma CLAUSE CLAUSES} \mid \texttt{close\_curly\_bracket} \\
&\quad \texttt{CLAUSE} \rightarrow \texttt{open\_curly\_bracket LITERAL LITERALS} \\
&\quad \texttt{LITERALS} \rightarrow \texttt{comma LITERAL LITERALS} \mid \texttt{close\_curly\_bracket} \\
&\quad \texttt{LITERAL} \rightarrow \texttt{not variable} \mid \texttt{variable}\}
\end{aligned}
\tag{2.5}
$$

This grammar was also an $LL(1)$ grammar, tested using the tool from [1].
The next step after defining both grammars is the implementation of the Lexer class and Parser Class for each grammar. The implementation for this was taken from [7] and slightly modifed to suit the languages defined here.

# Chapter 3

# The Main Menu

## 3.1 Functionality

The main menu provides three types of interaction. The user is allowed to choose a procedure, choose a syntax for a propositional formula, also referred to as the input language, and provide a way to input one or several propositional formulas. A screen shot of the main menu is shown below.

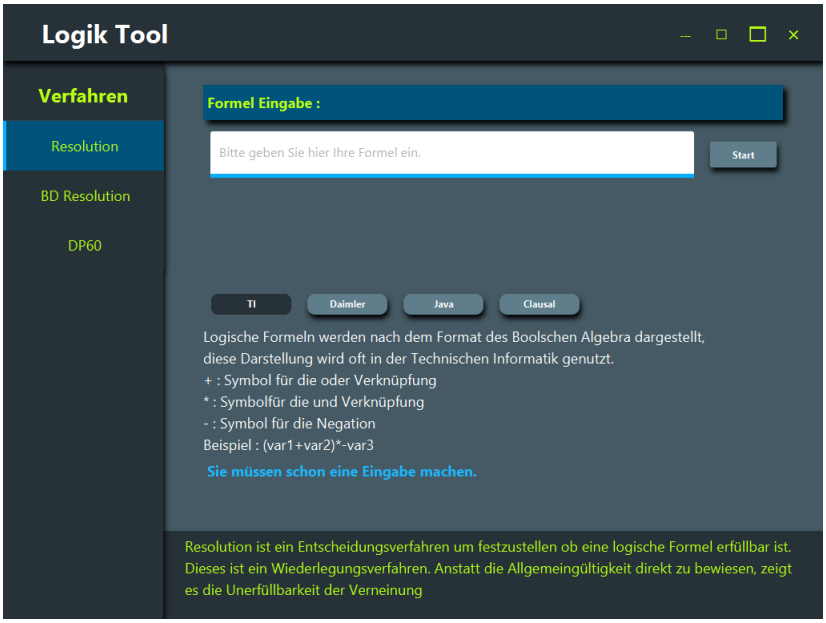


**Figure 3.1:** Main Menu

The following procedures were provided:

- **Resolution**: a refutation theorem proving technique.

- **Backward Dual Resolution**: a procedure to minimize a propositional formula in DNF form.
- **DP60**: this procedure will test the satisfiability of a propositional formula.

The choice of syntaxes were:

- **TI**: this syntax is often used in the computer engineering lectures to represent propositional formulas.
- **Daimler**: a syntax copyrighted by Daimler to represent propositional formulas.
- **Java**: a syntax often used in java for logical expressions.
- **Clausal**: a set based representation for a propositional formula in normal form.

If one of the TI, Daimler or Java syntaxes is chosen, then the input language describes which symbols are to be used to represent the logic operators `AND`, `OR` and `NEGATION` (see Table 3.1). The application of variables and curved braces remains unchanged for all input languages thus they will not be specified for each input language.
If the clausal syntax is used, then the formula should be provided in a normal form, either in **CNF** or **DNF** form, thus representing a set of sets (see Section 2.1.1).

For TI, Daimler or Java syntax, the operators are described in the following table.

| | `AND` | `OR` | `NEGATION` |
|---|---|---|---|
| TI | + | * | − |
| Daimler | + | / | − |
| Java | — | & | ! |

**Table 3.1:** Symbols representing the logic operators `AND`, `OR` and `NEGATION` in the syntaxes of TI, Daimler and Java.

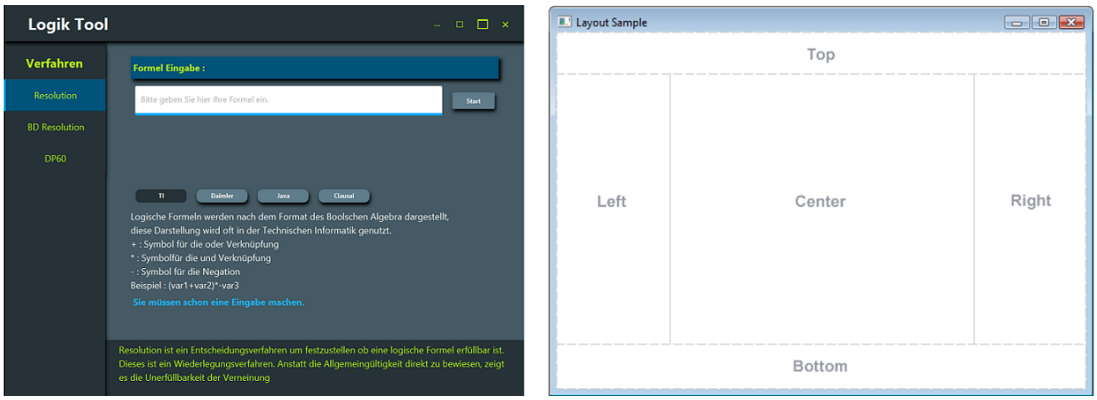


**Figure 3.2:**
**left**: The main menu of the application. An extra BorderPane was added into the center node.
**right** : A BorderPane is a JavaFX node containing 5 child nodes places on top, left, center, right and bottom of the node. The top and bottom children will automatically be resized to fill the width of the pane as well as preferred height. The children left, center and right will automatically be resized to the maximum height possible as well as preferred width.

Once a formula has been entered, and both a procedure and a syntax are chosen, then the algorithm is started. Prior to the procedure starting, the formula(s) were parsed and converted into a normal form. If more than one formula is entered, then these were combined.

## 3.2 Layout and Components

The layout chosen for the GUI after starting the application was `BorderPane` (see Figure 3.2).
The GUI size was $1024 \times 768$, which suffices most modern equipment. The design principles of this GUI were based on the description found in [2].
The first step was to create an elevation hierarchy for the JavaFX Nodes. Starting at this point the term JavaFX Node will be referred to as node. The elevation hierarchy can be described as follows: the top node is the highest element followed by the left node, then the bottom node and the lowest point is the center pane. The difference in elevation is made visible by the type of shadow each of the components cast on the lower placed components. The shadow changes based on how higher the component is supposed to be.
The second principle was to show all components even when their actions are not available yet. The application displays an error message if the action is triggered but the requirements for the action are not fulfilled.

The last step was that for each action there is a visible reaction. In the case of a button, e.g. it seems to be pulled towards the cursor when the cursor hovers over it. This impression was created by stretching the button and its shadow. Another example is when a Button is pressed, then the shadow disappears and the button is reduced in size. This was done by executing a translation on the y axis to the position of the cast shadow.

On the top node of the main menu, to the right, a JavaFX label was placed with the title "Logik Tools". From now on the term JavaFX Label will be referred to as label. The window controls were placed on the left side of the top node. This was necessary due to the customized borderless styling of the application. This hindered the resizing of the GUI using the edges, but positioning or dragging it anywhere on the screen is enabled.
The window controls consisted of labels that represent the behavior of a button. There were four labels which are described from left to right:

- **Minimize**: hides the GUI, it can be restored by a click on the application icon on the TaskBar.
- **Maximize**: resizes the GUI to fill up the screen with the TaskBar still visible.
- **Fullscreen**: fills the entire display with the GUI covering the TaskBar.
- **Close**: closes the application.

The left node contained selectable labels that can be toggled. Each label represented a procedure to be selected. The principle that each action should have a reaction was enforced in this case.
If a label was selected, the color of the label changed and a left border was added. Furthermore a short description of the procedure was displayed on the bottom of the BorderPane. When the cursor was positioned over a label, the color changed and the description of the label was shown on the bottom pane as long as the cursor hovered over it. When the cursor left the label without clicking on the information, the label returned to its previous state. This state could be the description of a previously selected procedure or just a reminder to select a procedure.

The center node provided two functionalities. The necessary components to input a formula and to start the selected procedure were positioned on top, consisting of a label, a JavaFX TextField and a JavaFX Button labeled **Start**, which was used for decorative purposes. Starting at this point the terms JavaFX Textfield and JavaFX Button will be referred to as textfield and button respectively. The textfield entered the formulas into the application. Furthermore if pressed, then the Enter key enabled multiple inputs. Each

propositional formula, which were given to the application and were accepted as a correct input were displayed underneath the textfield. It was possible to input propositional formulas in different input languages, since these would be automatically converted to the required normal form (set based form).
In order to calculate the input, it was necessary to select a procedure in advance, so the program knew which normal forms was required. It was important not to switch procedures between the propositional formulas, because not all procedures worked with the same normal form. Pressing the F2 key removes the last given formula from the multiple input list. Pressing the Start button will start the calculation for the chosen procedure. Should the Start Button be pressed but some necessary requirements were not fulfilled yet, then a customized message displayed the nature of the error.
On the lower part several buttons were provided to offer different syntaxes for the propositional formulas and a label component that would display a description of a syntax. These buttons were grouped together into a ToggleGroup. This grouping provided a variety of possibilities, which extended the behavior of a standard button. When a standard button was clicked, it remained in a pressed state, i.e. the way a button was drawn when clicked. If the button was clicked again, it returned to its default state. If another button was pressed in a ToggleGroup, then the button in a pressed state would be reset to its default state and the clicked button would be displayed in a pressed state. When a button in pressed state in this ToggleGroup was selected, then a simple description of the syntax for the format it represents would be displayed. When a cursor hovered over a different button in the ToggleGroup, the current description would be replaced by the actual description of this button. If this button was clicked, then the information would continue to be displayed even after the cursor left the button. If this button was not clicked, then the description would be restored to the state it was before the cursor entered that button.
Three labels were provided at the bottom of the center node, which were used to display erroneous inputs or missing choices. From the bottom up the labels are described as follows:

- The lowest label will display an error if no syntax was chosen for the formula.
- The label on the middle will display an error if no procedure was chosen.
- The top label will display an error if a syntax error is found in the input formula.

The bottom node displayed the information of a selected procedure, where the cursor was hovering over (see Section 3.1).

## 3.3 The Scene Graph

We will begin with a brief description of the components of a JavaFX application. A JavaFX application usually consists of three components, which are **the stage**, **the scene** and **the scene graph**. The stage, i.e. the window, is created by the platform the application is executed on. It contains a scene class.
The scene represents all physical contents of the JavaFX application, one of them being the contents of the scene graph.
The scene graph is a tree-like construction containing all nodes, visible and non visible, to be displayed on the application (see Figure 3.3). These nodes can be divided into two groups, the containers and the remaining nodes. The components in the containers group will be the branches on a scene graph. These nodes are designed to contain more nodes that are to be (or not to be) displayed on a specific layout inherent to the type of each container node, e.g. the container **HBox** will arrange the nodes horizontally and **VBox** will arrange them vertically. The remaining nodes are the leaves and usually they are going to be visible in the application.

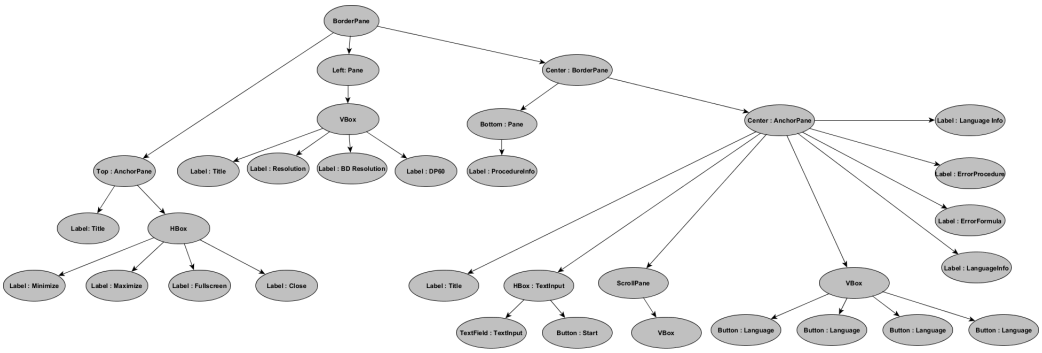


**Figure 3.3:** The scene graph corresponding to the main menu. The BorderPane at the top is the root node. This component will be passed to the scene.

The node located at the top of the scene graph is the root node. This is the node that will be added to the scene, together with all its child components. A BorderPane was chosen for the main menu, with the reasoning that each of the five nodes would wrap around or resize the component it would host. Only the top, left and center node were used in the main menu. The components in the top and left node had a fixed width and height respectively. Only the center node was enabled to be resized both horizontally and vertically.

The top node contained an **AnchorPane**, which allowed the edges of the children nodes to be anchored to an offset from the panes own edge, thus forcing the child to keep its relative position or resize depending on which

edges were anchored. The top pane displayed the title of the window and the window controls grouped in an HBox. The windows control group was anchored to the right, so that when the windows resizes it keeps the distance from the right edge of the application. Without this anchoring, the window control group would stay in the same position no matter how big the window was. The purpose of the HBox was not only to keep the window controls in the same position relative to the border, but to keep all four labels aligned to the same height and next to each other. Thus both **the HBox spacing**, i.e. the value that defines how much space is to be left in between the children, and **the HBox padding**, which is the value that sets how much space should be left between the border and the children, were both set to zero.

The left node contained a VBox, with a label for the title and 3 labels that functioned like a toggle box representing each a procedure to be chosen from, as its children. A bottom border was added to the title label in order to create a divider between the title and the procedures. The VBox spacing and padding were also set to zero.

The center node had a more complex structure and it contained most of the visible components. For the top component a BorderPane was used, in order to create the impression that the bottom pane was positioned under the left pane. This would not have been achieved by using a single BorderPane, since the botom node was stretched to the full width of the application. A VBox might have been able to fulfill the same purpose, since only the center and bottom nodes were used, but the BorderPane achieved the same result without any extra settings.
The bottom node of this BorderPane contained a simple pane with a label used to display the description of the selected procedure.
The more complex arrangement happened in the center node of the BorderPane, which again contained a AnchorPane. This AnchorPane was used to fix the left and right edges using fixed distance with respect to their corresponding borders. This caused the components to resize in width and changed the size of the AnchorPane too. Due to the fixed sizes of the respective neighboring components, only this Anchorpane would resize if the size of the application was changed. Using a VBox would seem to have achieved the same result with less work. The problem with a VBox is that not all JavaFX components have the same **fill** or **resize** behavior (see [3]).
The resize behavior refers to the ability of a component to automatically increase or decrease in size with the parent component. The fill behavior is the ability to use up all the free space available in the parent component. Components not having this ability will be resized by generating new code and binding it to one of the parent nodes.
A label with a fixed height was set at the top of the AnchorPane with text requesting an input. Below this label an HBox was positioned containing a textfield and a Start button, thus ensuring the textfield and button stay

aligned.
The button had a fixed size, thus creating a formula describing the width of the textfield and the width of HBox minus the space the button takes. Then resizing the application would resize the textfield and kept the button always in its relative position the edge of the application.
The upper edge of both labels containing the title and the HBox were anchored to the top of the AnchorPane. This way the position to the upper border was fixed, and they did not overlap with the other component which would resize in width and height. This component was a ScrollPane containing a VBox. The VBox arranged the given propositional formulas from top to bottom when the input had multiple lines. Should the size of the VBox exceed the viewport of the ScrollPane, then it would clip the view and add ScrollBars to the VBox. The ScrollPane edges were all anchored on the AnchorPane, thus forcing it to resize when the AnchorPane resizes.
On the lower part of the AnchorPane was an HBox and a label. The Hbox served as a container for dynamically generated buttons, ensuring the availability of different input languages. The label displayed the information of the input language.

Three labels were arranged vertically on the bottom, each displaying a certain type of error. The upper one displayed an error if there is no input or if there was a parser error. The middle one displayed an error if no procedure was chosen and the lower one if no language was chosen.
All the components described here were created using the scene builder, except for the labels contained in the ScrollPane, which described a multiple line input and the language buttons.

## 3.4 The Scene Builder

There are two ways to build a visual layout in JavaFX, one is by coding each component and implementing the scene graph manually. The other is by defining each component in an file using the FXML language. FXML is an XML based markup language used to define the structure of the scene graph. This can be done since the hierarchical structure of XML and the scene graph are similar. The biggest advantage of using FXML is that there is a visual editor named Scenebuilder, which facilitates the creation of a user interface and stores it directly as an FXML file, then Java will create the necessary instances from this FXML file. Because event handlers can be defined for a component, the implementation still has to be done the traditional way, thus it is not a completely codeless solution. A Controller class can be defined for the FXML file in which event handlers and other features that are not be defined by FXML, can be set or instantiated, e.g. the fill or resize behavior needs to be coded into the controller.

A way to build a visual layout was to let the FXML file define all the static components to the user interface, and let the controller generate additional content as was needed or set the settings that could not be accessed through the FXML file. In this application all static components were defined in the FXML File of their respective scene. Specifically for the Main Menu only the labels for a multiple line input and the buttons for the input languages were dynamically generated, the remaining components were all created using the Scenebuilder and stored in an FXML file.

Customizing the user interface components, which refered to color size and other behaviors, were not coded into the program but rather stored in a separate file that were added using Scenebuilder or by directly accessing the code on the FXML file. For this purpose JavaFX offers JavaFX Cascading Style Sheets (CSS), based on the W3C CCS version 2.1, and allows modifying most parameters of a node.
Of course it would have been possible to do this by programming, but this way offered the possibility to define colors and behavior as templates and apply them simultaneously to several components.

## 3.5 The Controller

All the calculation and logic were packed in to the Main_Menu_Controller class, instead of implementing a Model-View-Controller (MVC) pattern for the main menu. The resulting size of this class was a disadvantage, but most of the code was needed to set up the behavior of the UI.

Most of the fields defined in the Main_Menu_Controller class are references to UI components defined in the FXML file. The declaration of such fields are preceded by the tag @FXML, which informs JavaFX that the declaration of this field is done by the FXML file belonging to the controller. A list of the field is provided here:

- `root_node`: its main function is to provide a reference to its parent, the stage, allowing to switch to another scene once the input process is finished. Also since the application is borderless, extra functionality was needed to move the window. The root_node will implement the listeners that will enable dragging the window.

- `bottom_pane` and `pane_center` are references to the top containers on the center and the bottom of the UI.

- `label_resolution`, `label_bd_resolution` and `label_dp60` are the selectable labels that will determine which calculation procedure will be

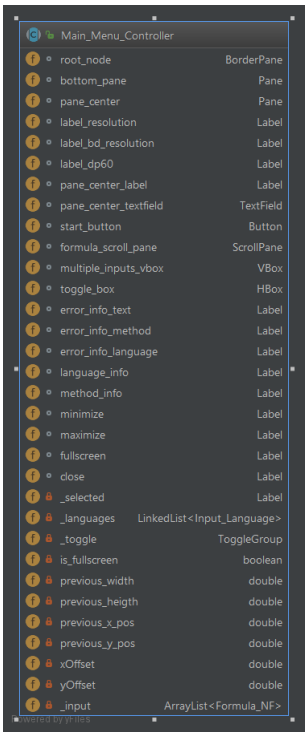


**Figure 3.4:** All fields of the main menu controller

run after.

- `pane_center_label`: the label positioned over the text field used to input the propositional formula.
- `pane_center_textfield`: a text field used to input the propositional formula.
- `start_button`: a button with the text Start, which will start the calculations when pressed.
- `formula_scroll_pane`: a scroll pane that will host the VBox multiple_inputs_vbox.
- `multiple_inputs_vbox`: if more than one propositional formula is to be used, all the already processed propositional formulas will be displayed here.
- `toggle_box`: an HBox hosting the input language buttons.
- `is_fullscreen`: a Boolean that represent if the application.
- `error_info_text`; `error_info_method` and `error_info_language` will display error messages for the text field, method selection and input language selection respectively.

- `language_info`: a label that will display a simple description of the selected input language.
- `method_info`: a label that will display a simple description of the selected calculation method.
- `minimize`; `maximize` and `fullscreen` are clickable labels controling the size of the stage.
- `close`: a clickable label that will end the application.

All the previously explained fields are the fields defined in the FXML file. The following fields are are not part of the FXML file but store information necessary for the Main_Menu class to function properly.

`_selected`: stores a reference to the chosen method.

`_languages`: stores the list of input languages.

`_toggle`: groups the input language buttons into one component allowing only one button to be selected.

`previous_width`; `previous_height`; `previous_x_pos` and `previous_y_pos`: store the original width and height of the application, as well as the last known position on the screen before the application was maximized or changed to full screen view.

`xOffset`, `yOffset`: store the last position before a drag event, allowing to calculate direction and distance of the movement of the stage.

`_input`: an array list that stores all given and processed formulas.

Here a description of what each method does:

- `public void initialize()` is called automatically by JavaFX. It sets up everything that was not set up via FXML. The methods `_info`, `_toggle`, `_input` and `is_fullscreen` are directly initialized, specifically for Main_Menu_Controller.
- `set_up_window_state(...)`: when the scene is changed by a calculation start or an calculation end, the new scene is to receive the original size and position of the stage.
- `buttonEventHandler(Event)`: an event handler for the start_button defined on the FXML file. It will start the change of scene if all requirements are met.

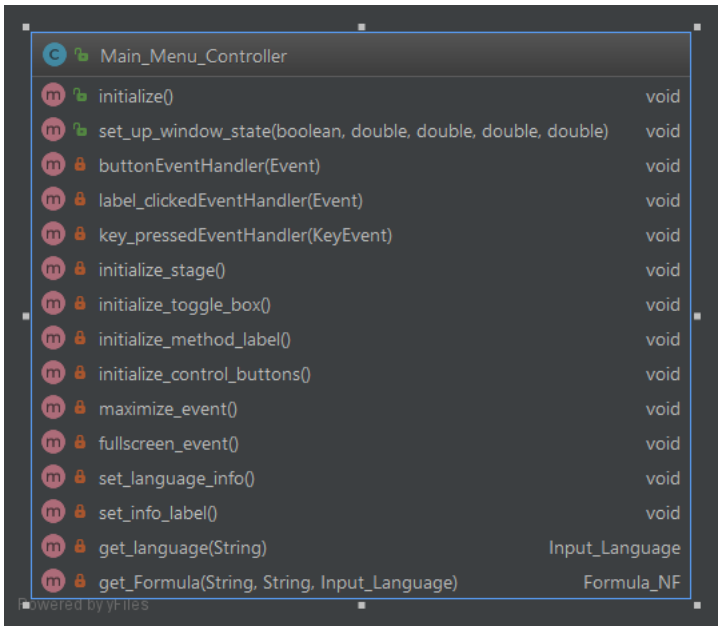


**Figure 3.5:** The Methods of the Main_Menu_Controller class

- `label_clickedEventHandler(Event)` is raised when a label representing one of the procedures is clicked.
- `key_pressedEventHandler(Event)` is raised every time a key is pressed while focused on the pane_center_text field.
- `initialize_stage()` sets the listeners to implement the dragging of the stage.
- `initialize_toggle_box()`: all available input languages will be loaded and the toggle_box will be populated with one button for each language.
- `initialize_method_label()` implements the functionality to display a description on the method_info while hovering over a label representing one of the available procedures.
- `initialize_control_buttons()` sets up the event handler for the labels minimize, maximize, fullscreen and close. It also records the original size of the stage.
- `maximize_event()` is raised when the maximized label is pressed.
- `fullscreen_event()` is raised when the fullscreen label is pressed.
- `set_language_info`: a method that gets the description of the selected input language or it loads a message in case none is selected.

- `set_info_label` sets the a short description of the selected procedure on the label method_info.

- `get_language(String)` retrieves a language with a given name. If no match is found, then a null reference is returned.

- `get_formula`(...) parses and converts a given string representing a propositional formula into a formula in normal form.

### 3.5.1 The Control Buttons

A closer look at the resize functionality of the stages is provided here. The applications stage was set to borderless, which removed the frame the operating system provided, including the minimize, maximize and close buttons offered by the operating system. Consequently this functionality was to be implemented separately into the applications stage as a set of labels called the control buttons.
The functionality of the control buttons was the same for all scenes. The group consisted of four labels, positioned from left to right (see Fig. 3.2 Left) as minimized, maximized, fullscreen and close. The operating system does not offer the fullscreen label as a control button on a frame, but since the application was intended to be used during a demonstration, it was deemed useful to add it as an extra option.
The method `initialize_control_buttons` was in charge of setting up the event handlers for those labels. The event handler for the minimize and close label comprised of one line each, a call to the respective java function that minimized or terminated the application respectively. Unfortunately the maximize and fullscreen java commands did not work as expected, caused by the borderless Stage.
In order to understand the problem, we will explain what happens if the operating system frame is kept. A call to `stage.setMaximized(true)` will cause the application to fill the display just leaving the systems taskbar visible. A call to `stage.setFullscreen(true)` will remove the operating system frame and make the application stage fill the display entirely. As a consequence a message is displayed for a couple of seconds stating which key to press to return to the application stage to its previous state.
In the case of a borderless Stage, a call to `stage.setMaximize(true)` caused the applications stage to fill the whole display covering the taskbar. It behaved as a call to `stage.fullscreen(true)` without the extra message on the key to leave the fullscreen state. This problem was solved by calculating the maximized size calling on the operating system available space for a maximized window and setting the results manually to the application stage. In order to be able to determine if the application was maximized, the field is_fullscreen

was introduced. The naming is a bit confusing, but using is_maximized would not have created a different effect, since a call to `stage.maximized(true)` was used to set the applications stage to fullscreen. The reasoning is the absence of the extra overlay with indicating the key needed to be pressed to return to the previous state. Both maximize and fullscreen labels, called the methods `maximized_event()` and `fullscreen_event()` respectively to handle the transitions of the stage into the expected state. A graph representing the transitions is described in Figure 3.6. Table 3.2 describes the transitions and the actions in more in detail.

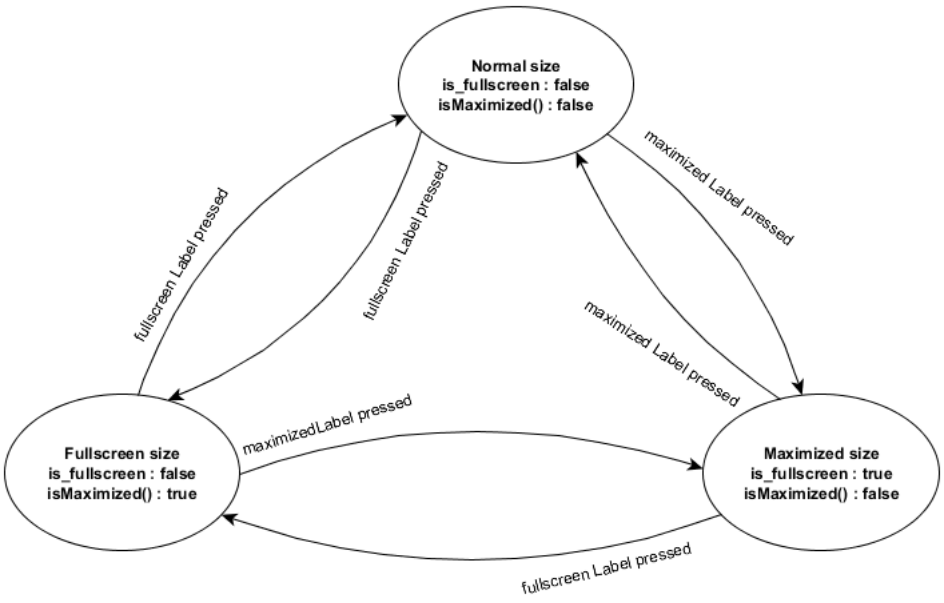


**Figure 3.6:** Different states of the application stage size and how each event changes them

### 3.5.2 The Procedure Labels

A set of three labels were positioned at the left side of the user interface. Their functions were to determine which procedure was to be applied to the given formula.
The clicked event handler of these labels called on the same method `label_clicked_EventHandler(Event)`, which saved a reference of the clicked label on the field selected. Also it added a short description of the selected method to the method_info Label at the bottom of the user interface.
If the mouse entered or left these labels or the labels were clicked, then all three labels had a change of state. These visual changes were defined in the respective css files. The event handlers for the mouse enter event and the mouse

| Actual state | Control pressed | Next state | Actions |
|---|---|---|---|
| **NORMAL SIZE** | maximize | Maximized size | Saves actual position into previous_x_pos and previous_y_pos<br>Calculates and sets application Stage to maximum size<br>is_fullscreen: true |
| **NORMAL SIZE** | fullscreen | Fullscreen size | Saves actual position into previous_x_pos and previous_y_pos<br>Sets application Stage to fullscreen |
| **MAXIMIZED SIZE** | maximize | Normal size | Resets to original size and position before the call to maximized<br>is_fullscreen: false |
| **MAXIMIZED SIZE** | fullscreen | Fullscreen size | Sets application Stage to fullscreen<br>is_fullscreen: false |
| **FULSCREEN SIZE** | maximize | Maximized size | Calculates and sets application Stage to maximum size<br>is_fullscreen: true |
| **FULSCREEN SIZE** | fullscreen | Normal size | Resets to original size and position before the call to maximized |

**Table 3.2:** Transitions of the stage and the respective actions

leave caused the selected procedure description to be temporarily overwriten with the one the mouse was hovering over.

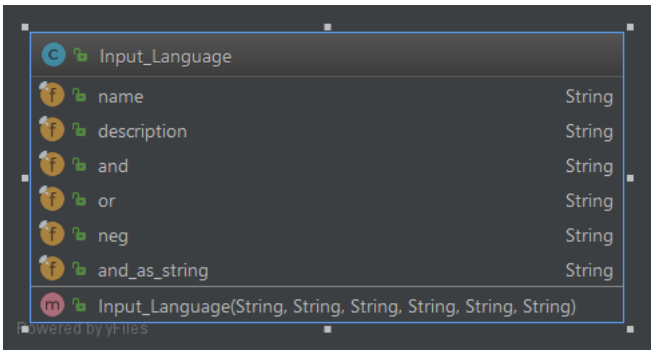


**Figure 3.7:** Input Language Class diagram

### 3.5.3 The Toggle Group

This control component was generated dynamically at runtime. It was populated through the method `initialize_toggle_box()` which added a button to it for each input language available.
The input languages were stored in a xml file. In order to be able to change

the parameters without having to sift through the code the following scheme was used.

```
<Language>
    <Name> </Name>
    <OR></OR>
    <AND></AND>
    <NOT></NOT>
    <Description> </Description>
    <DescriptionOR> </DescriptionOR>
    <DescriptionAND> </DescriptionAND>
    <DescriptionNOT> </DescriptionNOT>
    <DescriptionEx> </DescriptionEx>
    <ANDString></ANDString>
</Language>
```

The tag `<Name>` represents the string that will be displayed on the button added to the toggle box and will be used to identify the input language.
The tags `<OR>` , `<AND>` and `<NOT>` will store the representations of the respective operator. Later on this representation might differ from the string used for the operator.
The tags `<Description>` and `<DescriptionXXX>` contain some words of the input language and a reference to the used operators as well as an simple example of a propositional formula represented in each language. This information will be displayed on the user interface as the description of the language on the language_info Label.
`<ANDString>` contains the string for the `AND` operator.

The sole purpose of the XMLReader class was to scan through the languages and store them into a Input_Language class, functioning as a container for the information mentioned above.
The first step of the initialization was to read all the available input languages from the InputLanguages.xml file and create an Input_Language class for every single one of them.
The next step was to create a toggle button for each language and add it to the toggle box. When they were activated, the description of the input language would be displayed on the language_info panel. When the mouse pointer hovered over a toggle button, the language description would be replaced the actual description. The moment the mouse pointer left the toggle button the description was reverted to state it had before.
The toggle group provided a method `getSelectedToggle()` that returned selected toggle button as a node. If no toggle button was selected a nullpointer exception will be raised.

### 3.5.4 Key Pressed

After a thorough description of the main menu mechanics, we proceed to describe the two methods that start the calculations, i.e. `buttonEventHandler()` and `key_pressedEventHandler()`.

The functionality of passing multiple propositional formulas was provided by pressing the Enter key while focused on the text field. This started following algorithm:

> if the text field is empty, then display an error message and return
>
> if no procedure was selected, then display an error message and return
>
> get string from the text field
>
> get input_Language
>
> parse a given string into a formula
>
> if a parsing error is raised, then display error message and return
>
> convert a formula into the normal form needed for the procedure
>
> store the normal form in a list
>
> display the given string as a Label in multiple_inputs_vbox .

It is imperative that when using multiple inputs to not change the selected procedure, since this could affect the normal form being calculated.

### 3.5.5 The Start Button

The start button worked similarly to the key_pressedEventHandler. The main difference is that it allowed an empty text field, if the input list was not empty. In this case the input string would be parsed, converted and added to the input list but not displayed on the user interface, thus all formulas would be merged into one formula. The last step was to locate and load the scene corresponding to the chosen procedure.

# Chapter 4

# The Resolution Implementation

## 4.1 The ModelViewView Model

The Model-View-View model (MVVM) is a variation of the Model-View-Controller (MVC) pattern.
The MVC pattern is the first software architectural pattern used to implement user interfaces. As the name states it consists of three components:

- **Model** is the layer that represents the application's data.
- **View** is layer which usually represents the GUI.
- **Controller** is the layer containing the business logic of the application.

The view layer receives the user interaction and passes it to the controller. The controller assesses if the input is a valid one and passes the pertinent information to the model. If the data of the model is modified, then it notifies the view to update its content with the new data.

Further along, there was a need to separate responsibilities of the components, thus the Model-View-Presenter (MVP) pattern was created (see Figure 4.1 available online from [5]). In this case the user input went from the view layer to the presenter layer, replacing the previous controller layer, then the presenter manipulated the model. If data was changed, then the model notified the presenter, which then updated the information to the view. A consequence was that unit testing for presenter and the model was simplified. But the user interface (UI) logic was still embedded in the view layer, so to e.g. manage the data from presenter layer, the navigation logic or the UI state was still managed by the view layer.

The MVVM pattern takes the UI and puts it into the presenter layer, which is called the **View Model**, i.e. the view model now processes the data coming

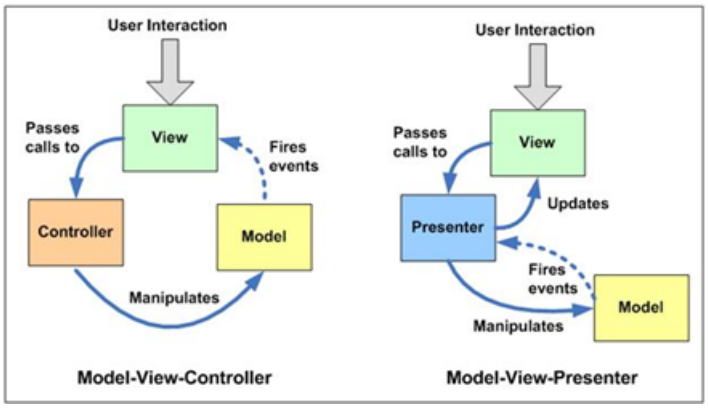


**Figure 4.1:**
**left:** Diagram of the interaction of the components of the MVC pattern.
**right:** Diagram for the interaction of the components of the MVP pattern.

from the model and serves the necessary information to the view, decoupling itself from the view. The MVVM serves the information different from the MVC or the MPV, where the information was passed to the View.
Since in the MVVM the view model has no reference to the view, but it has a reference to the view model, thus using data binders it connects to the information served. Any time the information served by the view model changes, then the view automatically updates its content.

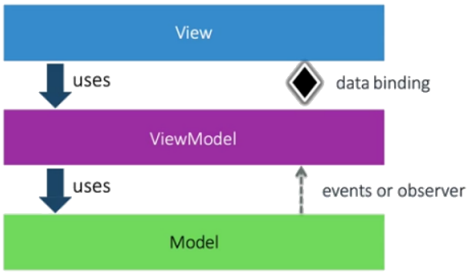


**Figure 4.2:** An example of the interaction of the components in the MVVM pattern.

When user input happens, passing the information was updated through binding too, but it is common to pass the information from the view to the view model through an event (see Figure 4.2 available online from [6]). The option of passing the information through an event was chosen for the application developed here.

Data binding in Java refers specifically to the use of properties. **Properties** are basically containers for a class and giving them the ability to raise events.

The events which are raised, consist mostly of changes of the state of the contained class, e.g. if a value is changed or updated or in the case of a list if an object was added or removed.
This means the view model offers the information data to the view using Public properties and the view binds its own properties to them. Binding two properties means that if one property changes in some way, then an event is raised. More specifically if the information data in the view model is altered, then the partner bound to it will be notified by an event.
For the partner to be able to process the event it needs an EventListener to handle the change. The interaction between ViewModel and Model is the same as in the MVP pattern.

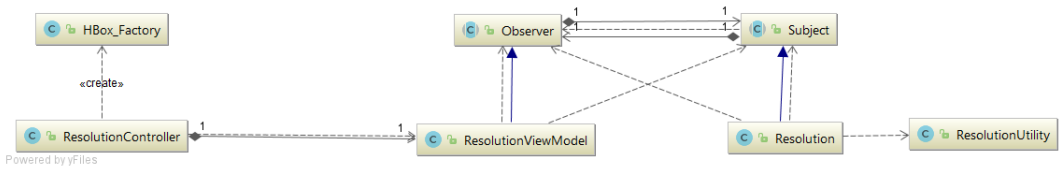


**Figure 4.3:** MVVM Pattern used for the Resolution procedure.

Basically the view is in charge of handling all components of the GUI, being a simple task if the data displayed is static. It tends to get more complicated, if the changes are dynamic, depending on how and where the new components are needed to display the new content, which is supposed to be generated.
One possibility is the view model, but this would imply that the view model will have to handle GUI components. Since it was stated that the view model was to serve the data, but it was not clearly stated if the GUI components were to be considered as data.
In this application the dynamically generated content was created and served by an extra class. As seen in Figure 4.3, here the view represented by the ResolutionController, took the information served by the ResolutionViewModel. The information dynamically generated which has no GUI component yet, was then passed to the HBox_Factory, returning a component that will be added to the GUI. This way there was no UI Logic in the view and no component was generated by the ViewModel. Futhermore the user input was passed to the ResolutionViewModel, which either processed the input itself or passed it to the model, in this case the Resolution Class. ResolutionViewModel and Resolution communicated through an observer pattern.

## 4.2 The Step Class and S_Calculation Class

These two abstract classes were used to store information needed by the view model or the view. The S_calculation class stores information about each calculation step, e.g. for the resolution it can store the information about a resolvent and its parents. On the other hand the step class groups the information corresponding to a step in the algorithm, e.g. calculating all resolvents.

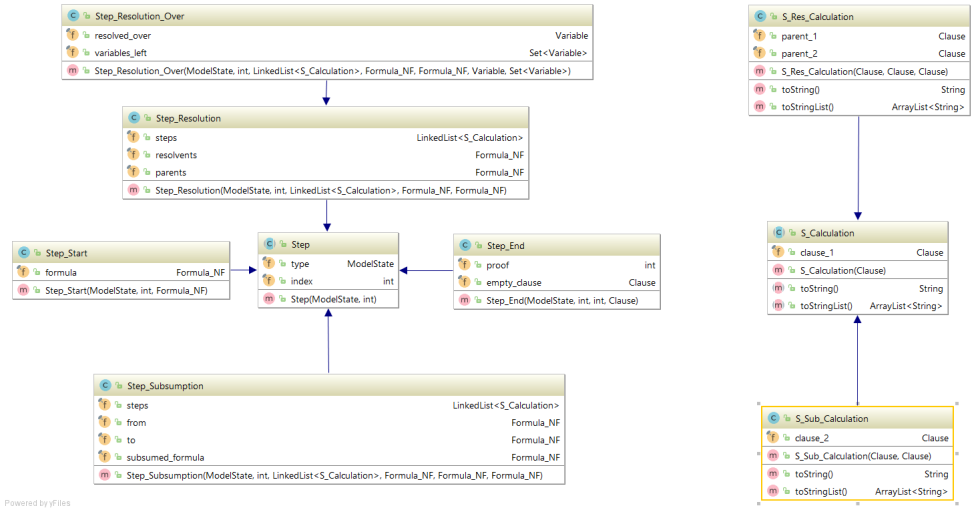


**Figure 4.4:**
Left Right: **left:** The class diagram for the Step Class and its subclasses used in the resolution algorithm.
**right:** The class diagram for S_Calculation and its subclasses used in the resolution algorithm.

The subclasses from S_Calculation used for the resolution algorithm are S_Res_Calculation and S_Sub_Calculation (see Figure 4.4).

S_Res_Calculation stored the information for a resolution step: the two clashing clauses are stored in parent_1 and parent_2 fields and the resolvent is stored in the clause_1 field.
The S_Sub_Calculation stored the information of a subsumption step, keeping the subsumed clause in the field clause_1 and the clause doing the subsumption in clause_2.
Both classes override the `toString` method, which was used mainly for testing purposes. A `toStringList` method was also implemented by both classes, returning a description of the calculation for the view. This description was returned as an ArrayList of string, which being lined up together formed the description. The reason to divide it, was so that the view can later format it

with different colors or fonts.

The step class, as it names states, represents the calculations and changes happening during a step of the algorithm. This refers specifically to all the calculations during an input event of the user of the application.
The subclasses used during the resolution algorithm were Step_Resolution, Step_Resolution_Over and Step_Subsumption (see Figure 4.4).
The abstract step class contained a ModelState field named type, which described for each step of the algorithm, which of the carried information corresponded to it. It also contained an integer field named index, stating the iteration of the algorithm.
Step_Resolution is the subclass that stored information about the resolution step. This information consisted of the formula the resolution step being applied and was stored in an instance of Formula_NF named parents.
The resulting resolvents were also stored in a Formula_NF instance named resolvents.
The last field, steps, is a LinkedList<S_Calculations> and stored every application of the resolution rule (see Definition 16).
The Step_Resolution_Over class stored the information corresponding to deriving all possible resolvents over a given variable.
Step_Resolution_Over is a subclass from Step_Resolution and stored additional information about the variable in the resolution rule it was applied on and the set of variables still contained in the resulting formula.

The set of variables left is of no interest for the resolution algorithm but for the DP60 algorithm. Step_Start and Step_End were also used. The Step_Start class stored no calculation, it just contained the given formula in set based representation. The Step_End class signaled the view that the algorithm has finished and contained the value of _proof and the empty clause if the formula was deemed unsatisfiable by the algorithm.

## 4.3 The Resolution Utility

The ResolutionUtility class is a class containing only static methods, thus when used it does not need to be instantiated. The main functionality of the this class was to provide the basic steps needed for the resolution algorithm (see section 2.1.2).

We will begin with the explanation of "derive all non trivial resolvents that have not been derived yet" in the resolution step. This basically means calculate $Res(F)^i$ in such a way that the resolvents calculated in $Res(F)^j$ for

all $j < i$ are not calculated again.
In order to achieve this, the following fact was used,

$$Res(F)^i = Res(F)^{i-1} \cup R^{i-1},$$

where $R^{i-1}$ are the resolvents of $Res(F)^{i-1}$ that have not been calculated yet.

To apply the resolution step for $Res(F)^i$, it was not necessary to derive resolvents over the clauses in $Res(F)^{i-1}$, since these were already in $R^{i-1}$.
New resolvents will be derived by calculating all possible resolvents over $R^{i-1}$, and all possible combinations of clashing clauses in between $Res(F)^{i-1}$ and $R^{i-1}$, i.e. all possible pairs of clashing clauses $C_1$ and $C_2$, such that $C_1 \in Res(F)^{i-1}$ and $C_2 \in R^{i-1}$.

The functionality needed to calculate the resolvents was offered by the ResolutionUtility class using two methods:

`resolution_over(Formula_NF` $r0$ `, Formula_NF` $r1$`, Variable` $a$`,`
`LinkedList<S_Calculation> steps)`

`resolution(Formula_NF` $r0$`, Formula_NF` $r1$`,`
`LinkedList<S_Calculation> steps)`

Both methods returned an instance of the Formula_NF class containing the resolvents and took two Formula_NF instances, $r0$ and $r1$ as arguments.
A resolvent was derived from a pair clashing clauses, where one clause is in $r0$ and the other in $r1$. This made it possible to calculate the resolvents over a set if $r0$ is equal to $r1$.

Also both methods took a LinkedList<S_Calculation> that stores all calculations made. This was necessary since a resolvent could be derived from more than one pair of clashing clauses, but only one resolvent with its respective parents would be added to the Formula. Also trivial clauses were not added to the formula but their information about their calculation was still stored.
So the LinkedList<S_Calculation> steps helped keep track of what was actually being calculated.
The `resolution_over` method calculates the resolvents for clashing clauses of a given variable, where as the resolution method iterates over all possible variables calling on the `resolution_over` method.

The `resolution_over` method first calculated all resolvents with clauses in $r0$ containing a positive literal over the variable $a$ and all clauses in $r1$ containing a negative literal over $a$.

In order to achieve this, it used the helper method `resolution_over (Formula_NF` $r0$ `, Formula_NF` $r1$`, Literal literal, LinkedList<S_Calculation> steps)`.
This method created a set of all clauses in $r0$ containing the given literal and another set of all clauses containing the clauses containing the complementary literal to the given literal. Then it calculated all resolvents by iterating through the clauses in both sets and calculating all possible resolvents. If a resolvent is a trivial clause then it will be discarded.
The resolvent was calculated by the method `resolution(Clause clause1, Clause clause2, Literal literal)`, taking two clashing clauses and the literal they clash on and returned the respective resolvent as defined in Definition 16 .

Another functionality in ResolutionUtility corresponded to the subsumption steps of the resolution algorithm. The method signature is

`Formula_NF subsumption(Formula_NF` $F$`, Formula_NF` $R$`, LinkedList<S_Calculation> steps)`.

This method removed all clauses from $F$ that are subsumed by a clause in $R$ and return the resulting formula $F$. This was done by iterating through all clauses of $F$ and looking for a clause that subsumes it in $R$. Like for the resolution this method took a LinkedList<S_Calculation> to store the executed subsumptions.

## 4.4 The Resolution Class

This class corresponds to the actual model in the MVVM pattern and it is the one in charge of executing the resolution algorithm, managing the results and passing the information to the view model. It also has the functionality to keep a log of the steps taken, so that the state of the resolution algorithm can be reverted to an older state.
We will begin with an overview of the fields contained in the resolution class:

`LinkedList<Step> _steps`: stores the resulting calculations of a executed step in order of appearance.

`int _proof`: When the algorithm end it will be set to 1 or 0 depending on the fact that the given formula is satisfiable or not, until that happens its value is -1.

`Formula_NF _formula` : stores $Res(F)^i$ as referred to it in the resolution algorithm in chapter 2.1.2.

`Formula_NF _resolventen`: stores $R^i$.

`Formula_NF _resolventen_temp`: stores $R^{i+1}$.

`Formula_NF _subsumed_resolvents`: stores $R^{i+1}$ after the forward subsumption step.

`Formula_NF _subsumed_formula`: stores $Res(F)^i$ after the backward subsumption step.

`ModelState _state`: stores the type of the step that is expected next.

`ModelState _back_step_state`: same as _state, it is used in the case of a step back event.

`Set<Variable> _formula_vars`: the set of variables in the actual formula.

`private boolean _fw`: `true` if forward subsumption was already executed, else `false`.

`private boolean _bw` : `true` if backward subsumption was already executed, else `false`.

`private int _index`: is a counter for the current iteration of the resolution algorithm.

`LinkedList<ResolutionLog> _logger`: keeps track of the state of the resolution algorithm and is used to change the step to a later state.

The Resolution class offers two public methods that are not getters for a field:

`attach(Observer)`: a method used to attach the view model to the resolution class.

`execute(EventObject)`: a method that receives an input from the view model, generally to execute a step for the resolution algorithm

### 4.4.1 The ModelState and the States

The ModelState is an enumeration containing different constants to determine state of an algorithm, i.e. the type of a Step instance or the type of an Event. The resolution class receives an EventObject from the view model, i.e. an instance of the ResolutionViewModel class, through the execute(EventObject) method.
EventObject is a Java class used to fire events and only contains information

about who created it. But the ModelEvent class was built using the EventObject as a base class. The ModelEvent class contains a ModelState field, informing the model, being the Resolution class, which step of the algorithm is to be executed next. It also contains a String field in case additional information is needed, e.g. in the case of the resolution_over(String) step.
Once an ObjectEvent is received, it is casted to a ModelEvent instance and based on the ModelState field, it decides which step is executed.
The following ModelState constants to which the execute method will react are:

`ModelState.RESOLUTION` which triggers the execute_resolution() method

`ModelState.RESOLUTION_OVER` which triggers the execute_resolution_over(String) method. Here the ModelEvent specifies the String field corresponding to a variable.

`ModelState.SUBSUMPTION_FW` which triggers the execute_forward_subsumption() method.

`ModelState.SUBSUMPTION_BW` which triggers the execute_backward_subsumption() method.

`ModelState.BACK` which triggers the execute_step_back() method.

Here is an overview of what these methods do:

`execute_resolution()`: This method calls upon the ResolutionUtility.resolution(...) to execute the resolution step. It also sets the state for the next step.

`execute_resolution_over(String)`: This method calls upon the ResolutionUtility.resolution_over(String) to execute the resolution step. It also sets the state for the next step.

`execute_forward_subsumption()`: This method calls upon the ResolutionUtility.subsumption(...) to execute the forward subsumption step. It also sets the state for the next step.

`execute_backward_subsumption()`: This method calls upon the ResolutionUtility.subsumption(...) to execute the backward resolution step. It also sets the state for the next step.

`execute_step_back()`: This method restores the state of the algorithm to the state it had before the last user input.

Before explaining in detail what each method does we will see how the _state field behaves.

The _state field controls if the method called by the execute(EventObject) actually does some calculation or just ends without doing anything.
An instance of the Resolution class will go through states similar to the steps of the resolution algorithm, i.e. the algorithm alternates in between a state for resolution, where only events for the resolution step are executed, and the subsumption state, where only events for forward and backward subsumption are accepted.
Contrary to the specification of the resolution algorithm, there is no fixed order for the subsumption steps, this means the backward resolution step can be called before the forward resolution. But each of the subsumption steps can only be called once. Only after both steps have been executed, then the state switches back to resolution.

The resolution step behavior is a bit more complex. For a an event containing `ModelState.RESOLUTION` as a type, the resolution step ends and it switches to the subsumption steps. But an event with type `ModelState.RESOLUTION_OVER` can be repeated as many times as elements are contained in `_formula_vars` before it will switch to the subsumption state.
The reason for it is that `resolution_over(String)` only calculates the resolvents for a given variable, so that process needs to be repeated until the resolvents over all variables are derived, although this is not necessary if an event of type `ModelState.RESOLUTION` is triggered. In this case the remaining resolvents are all calculated in one step and the states switches to subsumption.
For the resolution algorithm the values the _state field can take are the constant `ModelState.RESOLUTION` for the resolution step and `ModelState.SUBSUMPTION` for the subsumption steps. The third value the _state field can take is `ModelState.END`, which happens when the algorithm ends. This is caused by an empty clause derived or by empty resolvents.

### 4.4.2 The Resolution Step

The resolution algorithm implemented in the Resolution class calculated the resolvents as explained in section 4.3 , i.e. for $i > 0$ the resolvents for $Res(F)^{i+1}$ are calculated in function of $Res(F)^i$ and $R^i$.
$Res(F)^i$ and $R^i$ correspond to the fields _formula and _resolventen respectively where as $R^{i+1}$ corresponds to the field _resolventen_temp.
For the resolution step, there are two methods to calculate the resolvents, `execute_resolution()` that calculates all possible resolvents in one step

and `execute_resolution_over` that calculates all possible resolvent for the clashing clauses over a specified variable.

We will now take a closer look at the `execute_resolution()` method. The first step is to verify that the instance is in a resolution state, if this is not the case then the method returns without doing anything.
The next step is to iterate through the available variables and calculate the resolvents and store them in the field _resolventen_temp. Then the state of the calculation is saved by adding a new Step instance into the _steps field.
Now the _formula is updated to contain $Res(F)^{i+1}$ and $R^{i+1}$ is stored in _resolventen, since these are the fields the subsumption methods are going to work with.
Also for the fields _formula and _resolventen duplicate clauses are removed. Duplicate clauses refer to instances of the class Clause that represent the same clause without taking the parents into account.
Also the _state is set `ModelState.Subsumption` and _resolventen_temp is cleared before updating the state of the Resolution instance to the view model.

The last two steps consist of checking if the empty clause was derived or if no resolvent could be derived.
If an empty clause was derived then _state is set to `ModelState.End` and proof to 0, meaning the algorithm finished and the given formula is unsatisfiable. A last Step signaling the end of that calculation is added to _steps and the new state is updated to the view model.
In the case that _resolventen is empty, then the procedure is similar with the difference that proof is set to 1, i.e. the given formula is satisfiable. The last step is to create an log entry of the state for the whole instance of the Resolution class.

A call to execute_resolution_over(String var) is quite similar to execute_resolution(), the first difference is after checking the correctness of the state. Here the variable over which the resolvents are to be calculated is removed from _formula_vars. So basically _formula_vars keeps track of the variables over which resolvents are still to be derived.
It calculates the resolvents but only for the given variable and adds them into _resolventen_temp. If the resolution step consist of several events then _resolventen_temp will gather all the partial results, in spite of the Step added to the _steps field only containing the resolvents that were just derived.
The next step is to check if the empty clause was derived, then it will behave exactly like for the execute_resolution() method.
If there is no empty clause, then the next step is to verify if the resolution step is finished by checking if _formula_vars is empty.

If this is the case then the fields _state, _formula, _resolventen and _resolventen_temp are updated in exactly the same way as it was done for the resolution case.
The only difference is that an extra step will be added containing all resolvents without the duplicate clauses.
Then the view model is updated and tested if _resolventen is empty. If this is the case, then the calculations are ended in the same way it was done for the execute_resolution() method. The last step again is to generate a log entry.

### 4.4.3 The Subsumption Step

This contains both the forward subsumption as well as the backward subsumption. The forward subsumption is calculated through the method `execute_forward_subsumption()` which executes the following algorithm:

1. Check if the state is subsumption and forward subsumption has not been run before
2. Create a list to store the calculation
3. Calculate subsumes all clauses in _resolventen and store it in _subsumed_resolvents
4. Set _fw = `true`
5. Create a new Subsumption Step and store it in _steps
6. Update _resolventen = _subsumed_resolvents
7. If the subsumed resolvents are empty then end calculation

   else if backward subsumption has been already calculated then change state to resolution
8. Create log entry

Step 3 executes a call upon `ResolutionUtility.subsumption`(...) to calculate the subsumed formula.
In Step 4 setting _fw will inform the backward subsumption, that forward subsumption was already executed. Also the value of _fw is checked in step 1 to determine if the forward subsumption was previously run.
Step 6 updates the state of the Resolution instance to the view model.
Step 7 checks if the subsumption returned an empty set, if this is the case then no further calculations are allowed by setting _state to `ModelState.END`. The field _proof is set to 1 since an empty set implies a satisfiable formula. A new Step_End is added to _steps and the new state is then updated to the

view model.
If the result of the subsumption was not empty then it proceeds with step 8. Here it tests if forward subsumption was already calculated. If true then the state changes to the resolution state. The set of variables for the resolution is calculated and are stored in _formula_vars and the index is incremented to the next iteration.
The last step is to save the state of the instance as a log entry.

The backward subsumption works exactly in exactly the same way as the forward subsumption. The difference is in step 7, for the backward subsumption there is no proving if the subsumed formula is empty, thus the algorithm is as follows:

1. Check if the state is subsumption and forward subsumption has not been run before
2. Create a list to store the calculation
3. Calculate subsumes all clauses in _resolventen and store it in _subsumed_resolvents
4. Set _bw = `true`
5. Create a new Subsumption Step and store it in _steps
6. Update _resolventen = _subsumed_resolvents
7. If forward subsumption has been already calculated then change state to resolution
8. Create log entry

### 4.4.4 ResolutionState and update()

The Resolution class communicates with the view model through the `updated()` method by passing an instance of the `ResolutionState` class.
The `ResolutionState` class contains the information the view model needs to process the steps and offers the necessary information to the view. The values contained in this class are the state the calculations are in.
For the case that the state corresponds to the resolution, then the set of the available variables for the resolution are included too.
The values of _fw and _bw are needed in the case of the state being subsumption, so that the view can be informed about which options it can offer to the user for the subsumption step.
In case the calculation ends, it is also necessary to pass the information if the formula was satisfiable or not.

### 4.4.5 The Log Class

The application offered the option to revert on step of the calculations. For the model, in this case the Resolution class, to be able to revert its state to an earlier state, it needed to keep track of the states it went through.
The state of the model was saved in a Log class that stored the values of all field at the moment of its creation with the exception of the field `ModelState _back_step_state` since this is not relevant for the resolution algorithm. This is needed when updating the reverted state to the view model.
After the end of each step a log is added to the `LinkedList<Log> _logger`. This is done by calling the method generate_log, that creates a log instance and adds it to the list.

The method in charge of reverting the state of the model is `execute_step_back()`. This method replaces all the actual values of the fields in the model with the ones stored in the last entry of the `LinkedList _logger`. Only the value for the state of the calculation was initially set to `ModelState.BACK` and the actual state was stored in _back_step_state. This was necessary because the view model decides on what action to take based on the value of _state in the `ResolutionState` instance and _back_step_state will tell the view model what the actual state of the calculation is.
After the state was updated to the view model then _state is set to its correct value.

## 4.5 The Resolution View Model

The function of the view model is not only to pass user input to the model but also to manage and update the information offered to the view.
The fields used to synchronize the information to the view are the following:

- `_steps`: a list containing the information to be displayed by the view.
- `_resolution`: is a BooleanProperty bound to the button_resolution Button visibility property in the view.
  This means that the button shows in the view when the value is `true` and is hidden when its `false`.
- `_resolution_over`: is a BooleanProperty that is bound to the menu_resolution_over MenuButton visibility property in the view.
- `_forward`: is a BooleanProperty that is bound to the button_fw_subsumption button visibility property in the view.

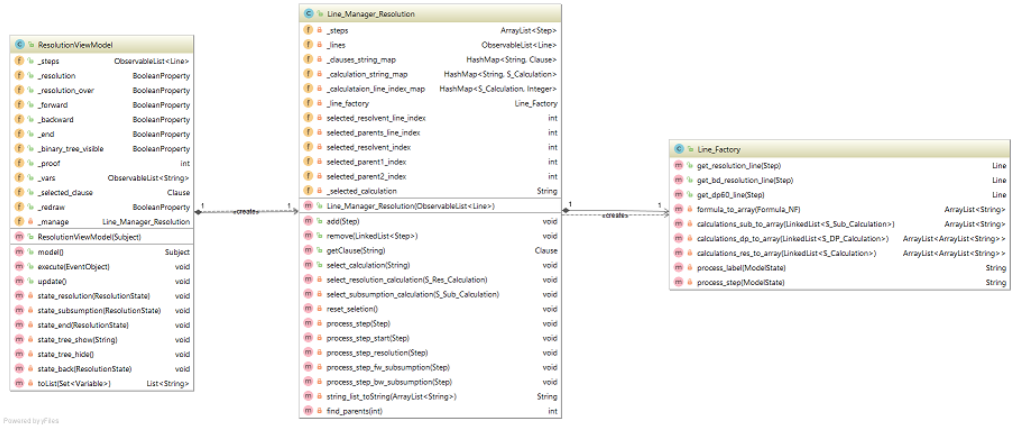


**Figure 4.5:** The class hierarchy of all the classes involved with the ResolutionViewModel

- `_backward`: is a BooleanProperty that is bound to the button_bw_subsumption button visibility property in the view.
- `_end`: is a BooleanProperty that is bound to the button_end button visibility property in the view.
- `_binary_tree_visible`: is a BooleanProperty that triggers the display of the deduction tree for a clause.
- `_vars`: is an VariableList containing String elements representing the variables over which resolvents can be derived.
  This list is displayed in the menu_resolution_over MenuButton.
- `_selected_clause`: is the clause that will have its deduction tree displayed.
- `_redraw`: triggers a repopulation of the panes containing information passed from the viewmodel.

The public methods that are offered and are not getters are:

- `execute(EventObject)`: a method used by the view to pass user input.
- `update()`: a method used by the model to update its state.

The update() method is the one in charge of setting up the information to be displayed and manages the visibility of the different buttons previously mentioned.
The workload of processing the newest Step instance from the model is given to a separate class, the `Line_Manager_Resolution class` (see Figure 4.5), for

which the ResolutionViewmodel class contains an instance called _manager. The manager refers to the field _steps so it can add the processed information for every new Step instance that comes from the model.
So basically the update() method sets all fields that are bound to the view except the _steps field. The _state field of the `ModelState` instance decides which actions to take and passes it to the viewmodel. These are as follows:

- If _state is `RESOLUTION` or `RESOLUTION_OVER`, then the buttons necessary to trigger a resolution step need to be visible on the view.
  This is done by setting the fields _resolution and _resolution_over to `true`.
  Also the buttons for the subsumption and `end` need to be hidden, by setting _forward, _backward and _end to `false`.
  Also _vars needs to be updated to the actual list of variables offered for resolution and the last Step instance is passed to _manager.

- If _state is `SUBSUMPTION` then the fields _resolution and _resolution_over are set to `false` in order to hide the respective buttons.
  The fields _forward and _backward are set based on the values passed in the ModelState instance.
  As seen in section 4.4 the values _fw and _bw are included in the ModelSate instance passed to the view model. These values state, if the forward subsumption and the backward subsumption were already executed.
  So to show the buttons of the subsumption steps, the negated value of _fw and _bw are assigned to _forward and _backward respectively.
  The _end field is also set to `false` and the _vars field is kept empty since it is not needed.
  Also the last Step instance is passed to _manager.

- If _state is `END` then the field _end is set to `true` and the fields corresponding to all other buttons are set to `false`. The _vars field is kept empty and the last Step instance is passed to _manager.

- If _state is `BACK` then the buttons are set, based on the value of the field _back_step_state containing the actual state of the model.
  The options are only `RESOLUTION`, `RESOLUTION_OVER` and `SUBSUMPTION` and these are handled for their respective cases.
  The content offered to display in the _steps field is emptied triggering the content of the view being emptied too. It is then repopulated by adding again each Step instance from the model.

The `execute(EventObject)` method takes input from the view, mostly this is input is predefined and passed on to the model. Only a request to show the deduction tree or to highlight calculations are passed to the _manager instance.

### 4.5.1 The Line_Manager_Resolution

This class extracts the information needed for the view from a Step instances and converts it in a format the view can use it.
This is done by using the `Line_Factory` class which offers of a method named `get_resolution_line(Step)`, that takes an step instances and returns the respective Line instance (see Figure 4.6).

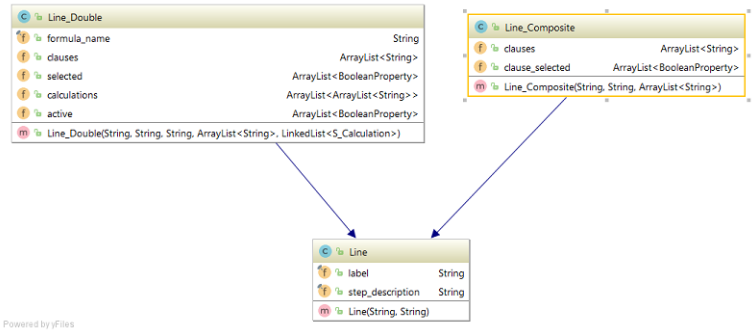


**Figure 4.6:** The hierarchy of the Line Classes used in the Resolution

The `Line_Factory` class creates a Line instance for a Step depending on the value of its field type as follows:

- For the value `START` a Line_Composite is generated.
- For the value `RESOLUTION`, `RESOLUTION_OVER`, `SUBSUMPTION_FW` and `SUBSUMPTION_BW` a Double_line is generated.
- For the value `END` a Line instance is generated.

For the Line the fields contain the following information:

- `label`: describes the name of the step.
- `step_description`: since the Line instance is only generated from the information of a Step_End instance, this field will contain the result of the calculation.

The Line_Composite fields contain following information:

- `label`: inherited from the Line class and contains the name of the step.
- `step_description`: Inherited from the Line class and contains the name of the displayed formula.

- `clauses`: a list containing the clauses representing the formula.
- `clause_selected`: a list consisting of BooleanProperty informing the view if a clause is supposed to be highlighted.
  This list has the same length as the list of clauses. An element with index $i$ containing the value `true` will cause the element in the field of clauses with the same index to be highlighted in the view.

The Line_Double fields contain the following information:

- `label`: inherited from the Line class and contains the name of the step.
- `step_description`: Inherited from the Line class and contains a short description of the step that was executed.
- `formula_name`: takes a String with the name of the formula contained in clauses.
- `clauses`: a list containing the clauses representing the formula.
- `selected`: a list consisting of BooleanProperty informing the view if a clause is supposed to be highlighted.
  This list has the same length as the list of clauses. An element with index $i$ containing the value `true` will cause the element in the field of clauses with the same index to be highlighted in the view.
- `calculations`: a list containing a description for each calculation executed during that step.
  Each description is divided into smaller Strings and stored as a list so that the view can format them separately.
- `active`: a list consisting of BooleanProperty informing the view if a clause is supposed to be "'grayed out"'. This list has the same length as the list of clauses. An element with index $i$ containing the value `false` will cause the element in the field of clauses with the same index to be grayed out in the view.
  This list is used exclusively by the Back Dual Resolution procedure.

When the method `add(Step)` is called by the ResolutionViewModel, then a new Line instance is generated for this Step instance by calling upon the method `get_resolution_line(Step)` from the LineFactory and added to the ObservableList<Line> _lines which is synchronized with the view.
The `add(Step)` method also makes a call to the private method `process_step(Step)`. With the call to this method the `Line_Manager_Resolution` manages and updates three Map instances which are needed to implement the functionality of displaying the deduction

tree or highlighting the clauses involved in a calculation.
These Maps are:

- `HasMap<String,Clause>` `_clauses_string_map`: this map ties the String representation of a clause to its respective instance.
  When the view requests a deduction tree for a clause, it passes the string representing the clause, then this map returns the respective instance.

- `HasMap<String,S_Calculation>` `_calculation_string_map`: this map ties the String representation of an S_Calculation to its respective instance.
  This is used when the view requests that the clauses involved in a calculation to be highlighted. These clauses are stored in the S_Calculation instances representing the requested calculation.

- `HasMap<S_Calculation,Integer>` `_calculation_line_index_map`: this map ties a calculation to the line where the calculation appeared.

Depending on the field type of the Step instance, information is added to one or all of the maps. For a Step where the field type has the value `END` nothing is done.
If the value of type is `Start`, only the clauses are added to _clauses_string_map since this Step does not contain any calculation.
For all the other Steps all three of the maps are updated.

The `method getClause(String)` returns the corresponding Clause instance from _clauses_string_map, if there is no match an empty clause is returned.

The last method used is the `select_calculations(String)`. This method takes a String representation of a calculation, finds the calculations and sets their `selected` or `clause_selected` value to `true`. Consequently the view will highlight those clauses.
In order to achieve this the first step is to find the S_Calculation instance corresponding to the String, which is stored in the _calculation_string_map. Then using _calculation_line_index_map it finds the Line instance where the calculation occurred.
Then it determines the index of the Lines instances stored in _lines containing the clause. This values are stored in the variables selected_resolvent_line and selected_parent_line. These names where chosen based on a resolution calculation but they hold the indexes for the Line instances of subsumed clauses, since the clauses involved in both calculations are always two.
The next step is to find the index of the clauses in their respective line and storing them in selected_resolvent_index, selected_parent1_index and

selected_parent2_index.
In the case of a subsumption calculation the parent2_index field is not used and set to -1.
All this information sets the value for highlighting the clauses. The reason for storing these values, is to be able to set them back to their original state, e.g. when another call select_calculation(String) happens.
Reverting to the old values is executed by the method `reset_selection()` setting the respective BooleanProperty values to `false` again.

## 4.6 The View

The view was composed of three Files: the Resolution_View.fxml file defining the static elements of the GUI, the Resolution_View.css file customizing the appearance and behavior of the elements in the GUI and the ResolutionController class that creating the connection with the ResolutionViewModel and initializing the behavior of the GUI by setting all the EventHandlers.
The structure of the view is similar to the one used in the MainMenu.
The root node was a BorderPane, but the left node was not used.

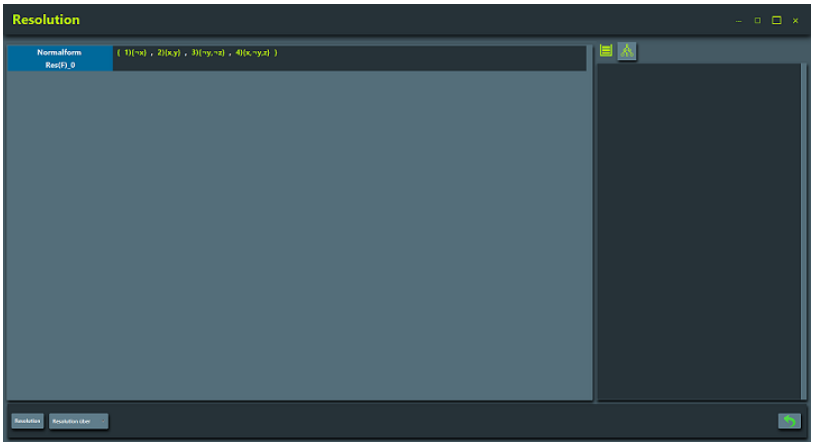


**Figure 4.7:** The GUI for the resolution procedure

The name of the procedure "Resolution" and the control buttons are in the top node. These buttons behave exactly the same as the buttons from the MainMenu. The center node is used to display information of each step of the algorithm and the right node displays the calculations for the steps displayed on the center node.
At the top of the node there are two buttons enabling to switch in between displaying the calculation and the deduction tree for a clause.

The bottom pane hosts the buttons that control the flow of the calculations. There are six buttons which are defined in the Resolution_View.FXML file. They are listed below based on their fxml id:

- `button_resolution`: is a Button labeled "Resolution". When pressed it calls on resolutionEventHandler(), that passes an EventObject of type `ModelState.RESOLUTION` to the viewmodel.

- `menu_resolution_over`: is a MenuButton labeled "Resolution over" containing a list of MenuItems corresponding to the variables for which a resolution step can be triggered.
  The content of the list is tied to the `Observable<List> _vars` that synchronizes its content with the viewmodel.
  When the content of _vars change an EventHandler is triggered updating the MenuItems.
  The EventHandler for _vars is implemented in the `initialize_variable_list()` method and creates a MenuItem for each element in _vars as well as it respective EventHandler which is triggered when the MenuItem is pressed. This EventHandler passes an EventObject of type `ModelState.RESOLUTION_OVER` to the viewmodel and the String of the chosen variable.

- `button_fw_subsumption`: is a Button labeled "Forward Subsumption". When pressed it calls on its EventHandler `fw_subusumptionEventHandler()` that passes an EventObject of type `ModelState.SUBUMPTION_FW` to the viewmodel.

- `button_bw_subsumption`: is a Button labeled "Backward Subsumption". When pressed it calls on its EventHandler `bw_subsumptionEventHandler()` that passes an EventObject of type `ModelState.SUBUMPTION_BW` to the viewmodel.

- `button_end`: is a Button labeled "END". When pressed it calls on the EventHandler `endEventHandler()` that switches the scene back to the MainMenu.

- `button_back`: is a Button labeled with a green arrow at the most right side of the bottom node. When pressed it calls on its EventHandler `back_EventHandler()` that passes an EventObject of type `ModelState.BACK` to the viewmodel.

The visibility properties for the previously mentioned buttons are bound to a BooleanProperty in the ResolutionViewModel, except for the `button_back`.

The center node displays the information for each step, which is synchronized through the `ObservableList<Line> _steps` that binds itself to the field `ObservableList<Line> _lines` in the ResolutionViewModel class. These lines are converted into a Node subclass that are added to Scene.
The problem was to comply with the MVVM pattern, since generating the the GUI elements falls under GUI logic, thus it should not be implemented in the View. But because the resulting content were instances of GUI elements, they could not be generated in the viewmodel either.
Usually the recommended solution is to let the viewmodel generate the content, but here another solution was chosen. The generating of the GUI elements from the _lines elements was done by an external class named HBox_Factory. This class offers a method `get_box(Line)` which takes a Line instance and returns a HBox instance with the desired content.
The HBox instance contained the name of Step, the name of the formula, a short description of the step and the resulting formula. Specifically for the Resolution procedure, the Hbox_Factory class offers a method `get_resolution_calculation(Line)` taking a line and returning a VBox instance containing a formatted view of the calculation corresponding to that Line instance that will be added to the right node of the BorderPane.

The EventHandler for the field _steps is initialized in the method `initialize_steps()`. When a new element is added then the EventHandler executes the following steps:

- The HBox for the new Line instance is generated and added to the center node.
- All the EventHandlers for the nodes representing a clause are generated, this EventHandler will make it possible to display the deduction tree when a clause is clicked on the GUI.
- If the Line instance contains calculations then a VBox containing Nodes representing the calculation are generated and added to left node and EventHandlers are also added. These EventHandlers will make it possible to highlight the clauses involved in their calculation.

When a Line is removed this means that a step back event was triggered and the contens of the center node and right node are going to be repopulated. When this happens the EventHandler will clear both nodes from any dynamic generated content.
The deduction tree for a clause is generated using the class Tree_Generator which offers a static method `generate_tree(Canvas,Clause)`. This method takes a Canvas instance and a Clause instance as arguments and will draw the deduction tree from that Clause instance into the given Canvas. The

Canvas used to draw the deduction tree is stored in the ResolutionController as binary_tree_view and the Clause is offered by the viewmodel a public field named _selected_clause.
The BooleanProperty _binary_tree_visible is bound the Boolean property of the same name in the viewmodel. When set to true its EventHandler triggers the display of the Canvas containing the deduction tree. The Canvas Node containing it, is initially displayed on the right node and will take the place of the calculations Node.
The buttons over the nodes make it possible to switch back and forth between the nodes. The option to stretch the deduction tree across the center of the GUI is enabled by clicking the button causing the right node of the border pane to be emptied and the center node to be filled with the deduction tree.
Clicking on the button will restore the original content of the center node of the BorderPane and move the deduction tree to the right node of the BorderPane.
Clicking on this button also restores the original content of the center node of the BorderPane and add the calculations to the right node of the BorderPane.

The EventHandler triggered when a clause is clicked on the GUI will refer the viewmodel to the _selected_clause field referencing the respective Clause instance and setting the _binary_tree_visible to `true`, causing the deduction tree to be displayed.

This summarizes the description of the implemented Resolution procedure. The description of the implementation for Back Dual Resolution and the DP60 procedure are generally similar to this procedure, differing mostly on the subclasses.

# Chapter 5

# The Back Dual Resolution Implementation

The structure of the Back Dual Resolution implementation (see Figure 5.1) is identical to the implementation for the resolution, since both were based on the MVVM pattern.

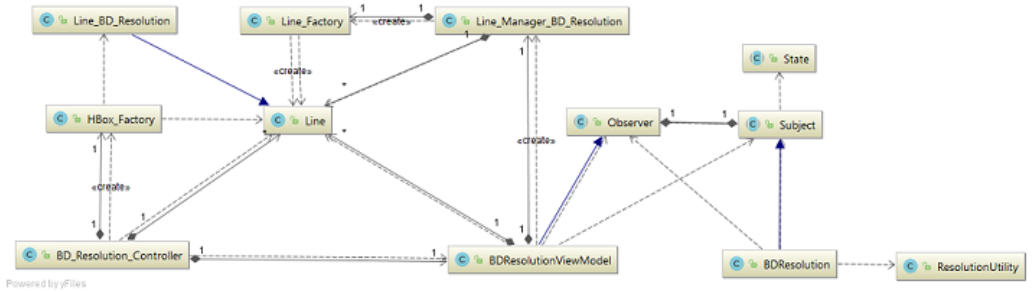


**Figure 5.1:** The class hierarchy for the Back Dual Resolution

## 5.1 The BDResolution class

This class is part of the model implementation of the MVVM pattern for the Back Dual Resolution. It is a subclass from the abstract class Subject used to implement the observer pattern together with the BDResolutionViewModel which is a subclass of the Observer class.
The BDResolution Class implements the Backward Dual Resolution algorithm described in section 2.1.2. The class constructor takes a instance of the Formula_NF class representing the propositional formula to be tested for satisfiability.

In order to execute the calculations, the ResolutionUtility class is used again because the derivation of resolvents is calculated the same way for clauses or terms. Same happens for the absorption rule than can be calculated with the implementation of the subsumption.
The execution order of the algorithm is controlled in the method `execute(EventObject)` taking an EventObject instance representing an action from the BDResolutionViewModel. The subclass ModelEvent of this EventObject instance has been described in the section 4.4.1.
Exactly like for the Resolution the ModelState _state field determines what is to be executed.

| **_state** | **Description** |
|---|---|
| `RESOLUTION` | Only lets a resolution step be executed |
| `BEFORE_SUBSUMPTION` | Tells the BDResolutionViewModel instance that its new state is going to be `SUBSUMPTION` |
| `SUBSUMPTION` | Only lets methods for the subsumption(absorption) to be executed. |
| `BEFORE_RESOLUTION` | Tells the BDResolutionViewModel instance that the new state is going to be `RESOLUTION`. |
| `END` | The algorithm is finished or more calculations can be done. |

**Table 5.1:** States the BDResolution class can take

The states change in following order:
`RESOLUTION` → `BEFORE_SUBSUMPTION` → `SUBSUMPTION` →`BEFORE_RESOLUTION` → `RESOLUTION`.
This loop repeats itself until the resolution no longer derives new resolvents. This is when the END state is set and the algorithm is finished.

The `SUBSUMPTION` state is comprised of two steps, the forward subsumption and the backward subsumption which can be executed in any order.
In order to be able to keep track of when the `SUBSUMPTION` state is finished, the BDResolution class has two fields _fw and _bw that keep track of whether the forward subsumption or the backward subsumption was executed. A `false` value means it has not been executed yet and `true` means it already has been executed.
Even if the steps are called subsumption, since it is executed with a formula in DNF, containing terms the operation is applied to, the formula is the absorption rule.

The states `END`, `BEFORE_SUBSUMPTION` and `BEFORE_RESOLUTION` are states the algorithm sets automatically and are not triggered by a user input.

The ModelState constants which are allowed to represent a user input in a ModelState instance are the following:

- `RESOLUTION`: triggers the execution of a resolution step by calling execute_resolution().
- `SUBSUMPTION_FW`: triggers the execution of a subsumption step calling execute_forward_subsumption().
- `SUBSUMPTION_BW`: triggers the execution of a subsumption step calling execute_backward_subsumption().

As the name states at the core of the algorithm is the resolution and like the resolution implementations the resolvents for $Res(F)^i$ is calculated in function of $Res(F)^{i-1}$ and $R^{i-1}$.
The BDResolution class executes this using an `ArrayList<Formula_NF> _formula` storing as a first element the propositional formula given by the user and the rest are the Resolvents for each step. This way the union of all elements up to an index $i$ represents $Res(F)^i$.

The method in charge of executing the resolution step is called `execute_resolution()` which determines the resolvent by deriving them from each element in _formula against its last element by calling up on the method `ResolutionUtility.resolution(_formula(`$i$`), _formula(last), LinkedList<Steps>)`.
The resolvents for each step are gathered in the field _resolventen. The resulting resolvent corresponding to the calculation for the resolvents was described in section 4.3.

The next step after calculating all the resolvents is to save all pertinent information for the calculation in the field LinkedList<Step> _steps. The BDResolution class uses the same implementation of Step as defined in section 4.2 with the difference that only two of the subclasses are used here, the Step_Resolution and the Step_Subsumption classes.
The Step subclass used to save the details for the calculations is a StepREsolution instance. It also makes an update of the results to the BDResolutionViewModel and it test if _resolventen is not empty, i.e. testing if no resolvents were derived. If this is the case then the algorithm ends by setting the field _state to `END`.
If there are new resolvents then the the BDResolution ViewModel is signaled to change states by setting the field _state to `BEFORE_SUBSUMPTION` and launches an update and finally sets the field _state to `SUBSUMPTION`.

The step applying the absorption rule to the new resolvents is executed by the method `execute_forward_subsumption()` and it iterates through all elements in the field _formula and it eliminates each of the absorbed terms in the field _resolventen and stores the resulting formula in field_subsumed_resolvents.
The result is saved as an instance of the Step_Subsumption class and updated to the BDResolutionViewModel. Furthermore the field _fw is set to `true`. Here it is tested again if the _subsumed_resolvents are empty and if this is true then the algorithm ends.

In the case that the field _subsumed_resolvents is not empty and the field _bw is `true` then the algorithm ends the `SUBSUMPTION` state by setting the field _state to `BEFORE_RESOLUTION`; resetting the values for _bw and _fw to `false` and adding _subsumed_resolvents to the field _formula.
Moreover _resolventen and _subsumed_formula are set to an empty instance of Formula_NF. After the update to the BDResolutionViewModel the value of _state is finally set to `RESOLUTION`. If none of the two cases are true, then the algorithm goes on in the `SUBSUMPTION` state waiting for the next user input.

The method `execute_backward_subsumption` applies the absorption rule to elements in _formula, which eliminates all terms absorbed by a term in the formula _resolventen.
If the field _fw is `true` then the `SUBSUMPTION` state ends the same way it was described for the `execute_forward_subsumption()` method. If _fw is `false`, then it keeps the same state.

The procedure of updating the state to the BDResolutionViewModel works exactly the same way it did for the Resolution implementation, it even uses the ResolutionState class for the update. The difference is that the field _vars is set to an empty set and _proof is 0. These values were not used in the BDResolution class.
The BDResolution class implements a `LinkedList<BDLog> _logger` that keeps track of the state to all fields and stores them in a BDLog class similar to the Resolution implementation. The difference here is that the BDLog contains the set of fields with which the BDResolution class works. The algorithm to set the state of the calculation back works in the same way as the one described for the Resolution in section 4.4.5.

## 5.2 The BDResolutionViewModel Class

Compared to the ResolutionViewModel the BDResolution view model is simpler, since it has less work to do, however the work flow is almost the same.

We will begin by describing the information offered to the view:

- `ObservableList<Line> _lines`: a list containing the information which is to be displayed in the view. For the BDResolution only instances of the subclass Line_Double were used.
- `_resolution`: a BooleanProperty controling if the Button that triggers the resolution step is visible or not.
- `_forward`: a BooleanProperty controling if the Button that triggers the absorption rule for the resolvents is visible or not.
- `_backward`: a BooleanProperty controling if the Button that triggers the absorption rule for the formula is visible or not.
- `_end`: a BooleanProperty controling when the End button is visible or not.

The fields _resolution, _forward, _backward and _end are connected through binding to their respective button, causing it to change visibility when the value is changed in the BDREsolutionViewModel class.

As for the Resolution implementation, when the BDResolution updates a new Step instance, this is passed to a class dedicated to processing its contents, generating a new Line instance and adding this to the field _lines.
The BDResolutionViewModel has a dedicated class for this purpose named Line_Manager_BD_Resolution, which was instantiated and referenced as the field named _line_manager.

The resolution takes input from the view through the method `execute(ObjectEvent)`, where the Object passed is actually an instance of a subclass from ObjectEvent called ModelEvent, as described in section 4.4.1. The method `execute(ObjectEvent)` casts the received Object to a ModelEvent instance.
If the contained field _state corresponds to `ModelState.SHOW_CALCULATIONS` then it passes the String representing a Calculation to _line_manager to make the clauses involved in the calculation visible.

When an `update()` call happens the BDResolutionViewModel either passes information to _line_manager and updates the visibility of the buttons on the view or takes a step back.
If the _state field from the ResolutionState instance obtained from the Resolution class equals `BEFORE_SUBSUMPTION` or `BEFORE_RESOLUTION` then it just sets the boolean properties _resolution, _forward, _backward and _end in accordance to the buttons needed for the state `SUBSUMPTION` and `RESOLUTION`

respectively.
If _state is equal to `END` then just the End button is displayed in the view and if _state is equal to `SUBSUMPTION` or `RESOLUTION` then the list with Steps is passed to the _line_manager updating the content offered to the view.
The last possible value for _state is `BACK`, in which case it behaves like in the case for the Resolution by clearing all contents out and repopulating the view with the Step instances passed from the BDResolution instance.

### 5.2.1 The Line Manager class

As for the Resolution implementation, the managing and updating of the information is outsourced into this class. We have mentioned at the beginning that only Line_Double instances are generated for the view, see section 4.5.1 for a detailed explanation.
Contrary to the resolution which displays each step in the view, only the resolvents generated in each resolution step are displayed in the BDResolution. This way each clause is only displayed only once in the view. So when absorbed the clause is not removed but "grayed out". Technically it is displayed but using a darker color.

The Line_Manager_BD_Resolution class contains three HashMap<> fields: _clauses_string_map; _calculations_string_map and _calculation_line_index_map. These HashMaps have the same purpose and functionality as the ones defined in the Line_Manager_Resolution class in section 4.5.1.

The BDResolutionViewModel adds a new Step instance through the method `add(Step)`. Depending on the field type of the Step different actions are taken.
If the field type is equal to `START` or `RESOLUTION` then the method `process_resolution(Step)` is called. This method generates a line instance from the step using the an instance of the Line_Factory class by using the method `get_bd_resolution_line(Step)`. This Step is of subclass Step_Resolution and the method handles two cases for it:

- The value of the type field is equal to `START`, then a Line_Double is created with the following arguments
  `("Start", "Formeln in Normalform", "R_"+Step_Resolution.index,`
  `Arraylist of the String representation of each clause,`
  `empty set of calculations)`

- The value of the type field is equal to `START`, then a Line_Double is created with the following arguments

```
("Resolution", "Alle nicht tautologische resolventen
von R_i , i                    <R_"+Step_Resolution.index,
"R_"+Step_Resolution.index, Arraylist of the String
representation of each clause, Step_Resolution.steps)
```

For each clause in the field _resolvents of the Step_Resolution instance an entry into clause_string_map is created and for each S_Calculation entries in _calculation_string_map and calculation_line_index_map are created.
Just like for the Resolution implementation, these maps were used to highlight the clauses involved in the calculations.

In the case that the field type is equal to `SUBSUMPTION_FW` the method `process_fw_subsumption(Step)` is called. Here the Step is of subclass Step_Subsumption. This value implies that clauses for the formula in the last line where absorbed.
For each S_Calculation instance in `Step_Subsumption.steps` the index of the absorbed clause is retrieved and the value with the same index in the active field of that line is set to `false`. This will inform the View to gray out that clause.
Also, for each calculation an entry in both _calculation_string_map and calculation_line_index_map is made and a Line instance is generated and added to _lines. The purpose of this line is to display the calculations that happened during that step.

In the case that the field type is equal to `SUBSUMPTION_BW` the method `process_bw_subsumption` is called. This value implies that clauses for all the lines but the last one were absorbed. For each S_Calculation instance in `Step_Subsumption.steps` the absorbed clause is retrieved by iterating through the Lines.
Once the clause is found, the the respective value on the field active is set to `false`. Entries for calculation_String_map and calculation_line_index_map are handled like for the `SUBSUMPTION_FW` case.

In order to highlight a calculation on the view, Line instances in _lines are manipulated. This is done in similar way as for the Resolution implementation.
The fields selected_resolvent_line_index; selected_parent1_line_index; selected_parent2_line_index; selected_resolvent_index; selected_parent1_index and selected_parent2_index store the index of the line and the index of the clause in the line for the clauses involved in a resolution calculation.
For a subsumption calculation the selected_resolvent_index and selected_resolvent_line_index are not used.

If the clauses involved in a resolution calculation need to be highlighted then the method `select_resolution_calculation(S_Res_Calculation)` is called. This method iterates through all the Line instances in _lines and saves the line index in its respective field.
If no clause is found, then the value is set to -1. This determines the index of the clause in the Line and sets its corresponding selected value to `true`.
In order to remove the highlight the method `reset` is called, this works exactly the same way as the one in the Resolution implementation.

## 5.3 The View

The Layout and static content of the GUI defined for the Back Dual Resolution is exactly the same as the one for the Resolution with two differences.
Since the calculation for the resolution over a variable was not included in the Back Dual Resolution, the MenuButton was not needed. The same goes for the whole functionality containing the creating and displaying of a deduction tree. The button to display a deduction tree and the ability to click on clause to display a deduction were absent in the implementation of the View.
The static content of the view was defined in the file BD_Resolution_View.fxml and the styling of its components was defined in the file BD_Resolution_View.css.

The content was displayed in the center node containing the resolvents and the right pane contained all the calculations. In order to create the content, an instance of HBox_Factory was used and the method `get_box(Line)` was used to generate HBox instances which were added to the center node.
This HBox was divided in two sides, a shorter left side containing two lines: the first one displaying the name of the step and the second displaying the name of the formula.
The right side also contained two lines, the first one displaying a short description of the step and the second displaying a formula in set based representation.
The container for the formula is an instance of a FlowBox that can grow to accommodate its content, in this case the growth was vertical.
The VBox containing the list of calculations was generated by the method `get_resolution_calculations(Line)` and is the same one used for the Resolution.

The controller tied to the View is the BD_Resolution_Controller class, where EventHanders and content were defined.
The EventHandlers for the Buttons were to generate user input in the bottom node and the control buttons on the top node were identical to the ones used

in the Resolution.

The main difference is how an update the `ObservableList<Line> _steps` was handled. The field _steps binds itself to the field _lines in the BDResolutionViewModel and triggers an EventHanlder every time the content in the List changes.
The EventHandler was implemented in the method `initialize_steps()` and it controls the properties of the new content and generating it. The Line subclass added to _steps is an instance of Line_Double and depending on the field label of the Line, the content generated was decided as follows:

- If the field label is equal to "'Resolution"' then the Line_Factory is used to generate an HBox to display the resolvents calculated in that step.
  This newly generated HBox is added to the center node.
  Then the Line_factory is used generate a VBox containing the calculation done during the resolution and each node representing a calculation is provided with an EventHandler.
  This EventHandler is a OnMouseClicked EventHandler that notifies the BResolutionViewmodel which calculations was clicked.

- If the field label is not equal to "'Resolution"' then the information in the Line instance corresponds to the one of an absorption, for which only calculations are displayed.

When Clauses are absorbed, then the content that is displayed changes, making it necessary to repeat the previously mentioned steps for every Line in _steps. This is solved by clearing the content and generating it again every single time the EventHandler for _steps is triggered.

# Chapter 6

# The Davis Putnam Algorithm

The last procedure implemented is the original Davis Putnam algorithm from 1960 (DP60). The structure was based on the MVVM pattern (see Fig. 6.1).

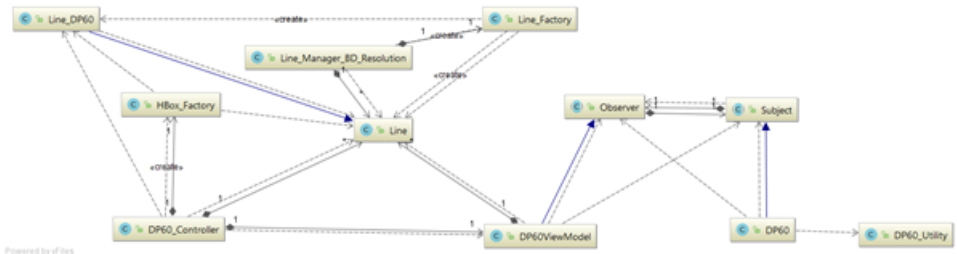


**Figure 6.1:** The class hierarchy of the DP60 procedure.

## 6.1 The DP60_Utility class

At the core of the calculations is the DP60_Utility class which implements the three rules defined for the algorithm introduced in section 2.1.2. The following methods offer the functionality to calculate these three rules:

- `rule_01(Formula_NF,Clause,LinkedList<S_DP_Calculation>)`:
  This method implements the Unit Literal Rule by removing all clauses from a Formula_NF instance containing the Literal from the Unit Clause. This unit clause contains its complementary Literal, thus this Literal is removed, but the clause itself is kept.
  Each step is documented as a S_DP_Calculation instance and passed to the caller using the LinkedList passed as an argument.

- `rule_02(Formula_NF, Literal, LinkedList<S_DP_Calculations>)`:
  This method implements the Pure Literal Rule by removing all clauses

from a Formula_NF that contains the Literal.
As in the `rule_01(...)` method all the steps are stored in the LinkedList.

- `rule_03(Formula_NF, Variable, LinkedList<S_Calculations>)`:
  This method implements the variable elimination by adding all resolvents over a variable to the Formula_NF instance and removing their parents.
  Calculating the resolvents is executed by using the ResolutionUtility class, specifically using the method `resolution_over(...)`.
  The steps the ResolutionUtility returns are all instances of S_Res_Calculation and are passed to the caller through the provided LinkedList.

Other than the functionality to calculate the three rules used in the DP60 algorithm, the DP60_Utility class offers three other methods:

- `get_literals(Formula_NF)`: This method retrieves all literals present in the Formula_NF instance.

- `get_pure_literals(Formula_NF, Set<Literal>)`: This method finds all pure literals present in the Formula_NF instance.
  In order to achieve this, it uses the method `get_literals(...)` to get all the literals in the Formula_NF and then keeps the ones for which the complementary literal is not present.
  The pure literals are stored in Set<Literals> and passed as an argument to the method. It returns the value `true` if a pure literal was found and `false` if no pure literal is present in the Formula_NF instance.

- `is_contained(Formula_NF,Clause)`: This method tests if a Clause is part of a Formula_NF instance.
  The difference to the method `contains_clause(Clause)` is that it compares only the literals contained in Clause. The one offered by the Formula_NF class uses equals to determine if the Clause is contained.

## 6.2 The DP60 class

Unlike the implementation for the Resolution and the Back Dual Resolution, the class in charge of implementing the algorithm is not State based. The execution of the user input coming from the DP60ViewModel has no specific order of execution.
In order to implement the DP60 algorithm, the DP60 class keeps three sets with information specific for each rule:

- `HashSet<Clause> _rule_01_clauses`: contains all the unit clauses which the Unit Literal rule can be applied to.

- `HashSet<Literal> _rule_02_pure_literals`: contains all the literals the Pure Literal rule can be applied to.

- `HashSet<Variables> _rul_03_variables`: contains all variables which can be eliminated using the Variable Elimination Rule.

The method `update_state()` is in charge of keeping these values updated.

The DP60 uses an instance of Step_DP to store the pertinent information corresponding to the execution of one of the rules. A Step_DP instance is a sub class of Step and stores a ModelState field describing which step was executed; the Formula the step was applied to together with the resulting formula; a LinkedList<Literal> storing information about the literals involved in the calculation together with two LinkedLists: one containing S_DP_Calculation instances and the other only S_Calculations.
The first is used to store the calculations resulting from executing rule01 and rule02, whereas the second stores the calculations resulting from executing rule03.
The Step_DP also provides a constructor for the results of each rule. All the Steps created by rule are stored in LinkedList<Step>_steps.

The field int _proof stores the result of the algorithm, being 0 if the given formula is unsatisfiable and 1 if the formula is satisfiable. If the satisfiability of the given formula is not yet decided, then the value of _proof is -1.

The ModelState field _state stores the corresponding value of the last executed state. It informs the DP60ViewModel which step was executed last when the DP60 class updated its state. The _back_step_state field function is the same as the one explained for the Resolution implementation.

When a calculation is done and the new Step instance is stored, then an update to the DP60ViewModel is triggered. The class containing the state is the State_DP class and it contains the fields _state and _back_step_state.
It also contains the fields: _steps, _rule_01_clauses; rule_02_clause and rule_03_variables. These fields were displayed in the View as options to create user input. The last field included was the field _proof.

The DP60 being a subclass from the Subject class enforces the Observer pattern, so it only communicates with the DP60ViewModel through the method `update()` and receives information through the method `execute(EventObject)`. The implementation of these methods are the same as in the procedures before.

The following inputs, referring to the field _state in the EventObject instance, for which the `execute(EventObject)` method be executed:

- `RULE_01_OVER`: For this value the method `execute_rule_01_over(String)` is called, where the String represents the Unit Clause.
  Using the method `get_unit_clause(String)` the respective Clause instance is retrieved and using DP60_Utility `rule_01(...)` method the Unit Clause Rule is applied to the actual formula stores in the field _formula. Then the result is stored in the field _result. A new Step_DP instance is generated and stored and the field _state is updated as well. Before updating the state of the class, the DP60ViewModel `update_state()` is called.
  The last two steps are calls, first to the method `test_end()` that will check if there is an empty clause or an empty formula and end the algorithm by setting a value for _proof and the value of _state to END. The last step is a call to the method `generate_log` which keeps a log of the state of the class.

- `RULE_01`: calls on the method `execute_rule_01()` taking one Clause contained in _rule_01_clauses and makes a call to `execute_rule_01_over(...)`.
  This is repeated until _rule_01_clauses is empty. It is important to mention that with each call to this method the content of _rule_01_clauses will change.

- `RULE_02_OVER`: For this value the method `execute_rule_02_over(String)` is called. The steps followed in the method are very similar to the ones used in `_rule_01_over(String)`.
  First instance of the pure literal is retrieved using the method `get_literal(String)`. With this literal the Pure literal Rule is applied to the actual formula using the DP_Utility method `rule_02over(...)` and the resulting formula is stored in the field _result.
  The rest of the steps are the same as for the `_rule_01_over(String)`.

- `RULE_02`: Here the method `execute_rule_02()` is called. This method retrieves all pure literals contained in _formula and iterates through them using a call to the method `_rule_02_over(String)`.
  This process is repeated until no more pure literals can be retrieved.

- `RULE_03`: For this value the method `execute_rule_03(String)` is called. This method takes a String with the name of variable and creates an instance of the class Variable with this String.

Then the method `rule_03(...)` from the DP_Utility is executed to eliminate the Variable and stores the result in the field _results. The rest of the steps are exactly the same as the ones in `_rule_01_over(String)`.

The DP60 class uses a DP60_Log class to store its state of the class. This class contains all fields except the LinkedList<DP60_Log> _logger field containing all logs. The method `generate_log()` creates an instance of DP60_Log and stores it in the list _logger.
When the method textttexecute(EventObject) receives an EventObject with _state equals to `BACK` then it calls the `execute_step_back()` method setting the state of the class to last state in logger.
The method works exactly in the same way as explained for the Resolution implementation.

## 6.3 The DP60ViewModel Class

As in the Resolution and Back Dual Resolution, the viewmodel is a subclass of Observer and offers properties to the view, thus used to synchronize information. The fields offered by the DP60ViewModel are:

- `ObservableList<Line> lines`: stores the information for each step that will be displayed in the View.
- `ObservableList<String> rule_01`: is a list containing a string representation of the unit clauses the Unit Literal step can be applied on.
- `ObservableList<String> rule_02`: is a list containing a string representation of the literal the Pure Literal step can be applied on.
- `ObservableList<String> rule_03`: is a list containing the names of each variable available for the Variable Elimination step.
- `BooleanProperty rule_1_visible`: is the property that is bound to the visible property of the Button and the MenuButton for the Unit Literal step.
- `BooleanProperty rule_2_visible`: is the property that is bound to the visible property of the Button and the MenuButton for the Pure Literal step.
- `BooleanProperty rule_3_visible`: is the property that is bound to the visible property of the MenuButton of the Variable Elimination step.
- `BooleanProperty end`: is the property that is bound to the visible property of the End Button. It tells the View when to show the Button.

- `BooleanProperty _redraw`: signals the view when the content it displays needs to be refreshed.

The only private field this class has is an instance of the Line_Manager_DP60 named _line_manager fulfilling the same functions as the Line_Manager_BD_Resolution or the Line_Manager_Resolution explained in preceding chapters.
In order to serve the content to the view only one subclass of Line is used, this is the Line_DP60 class.
This class receives information from the View through the `execute(EventModel)` Object and from the DP60 class through the `update()` method.
When execute(EventModel) is called by the view, this means that either there is user input to calculate a new step or to highlight a calculation.
If the field state of the model is equal to `SHOW_CALCULATIONS` then the information about the calculation is passed to the line manager to update the _line with the highlighted components. Then the value of the field _redraw is set to `true` so that the View updates the changes.

When a DP60 updates its state, it signals the DP60ViewModel then the `update()` method gets an instance of State_DP. If the value of the field state is `BACK` then _lines is cleared and all Steps are sent to a new instance of Line_Manager_DP, causing the view to clear all content and repopulate again.
If the value of the field state is not `BACK` then the last Step instance in the list steps is sent to _line_manager using the method `add(Step)`.
If the algorithm ended then the End button will be set to visible.
Finally the values for `rule_01`, `rule_02` and `rule_03` as well as for `rule_1_visible`, `rule_2_visible` and `rule_3_visible` are updated with the ones contained in the State_DP instance.

### 6.3.1 The Line_Manager_DP Class

The functionality this class offers is exactly the same as the Line_Manager_BD_Resolution, generating and adding a new instance of Line to the list _lines each time a new Step is added by the DP60ViewModel.
It also maintains the fields: HashMap<String, S_Calculation> _calculation_string_map and HashMap<S_Calculation, Integer> _calculation_line_index_map associating a calculation with its String representation and with the Line where it appears.
The Line is an element of the list _lines so the calculation is paired with the index this line has in the list.

This class also keeps the following four integer fields containing information about the selected clauses:

- `_calculation_line_index`: this field contains an index of the Line where the calculation occurred.
- `_clause_1_index`; `_clause_2_index`, `_clause_3_index`: these fields contain the indices the clause has in the field clauses of the respective line.

When a new Step is added by the DP60ViewModel through the `add(Step)` method, a new Line is generated using the Line_Factory by the method `get_dp60_line(Step)`. This line is then added to the list _lines.
Then for the calculation in the step, the entries in _calculation_string_map and _calculation_line_index_map are generated.

When a calculation needs to be highlighted, the view calls on the method `select_calculation(String)`retrieving the S_Calculation instance corresponding to the given string.
In function of the field type of the S_Calculation instance, the _line_manager updates the corresponding lines to highlight the clauses involved in the calculation. The `reset()` method sets selected clauses stored in the line_manage back to its original state.

## 6.4 The View

The fields involved in the view are mostly identical to the ones used in the Back Dual Resolution in section 5.3. One difference is the Buttons on the bottom pane and these are:

- `Button button_rule_1`: sends an EventObject of type RULE_01 to the DP60ViewModel.
- `MenuButton menu_rule_1`: sends an EventObject of type RULE_01_OVER to the DP60ViewModel.
- `Button button_rule_2`: sends an EventObject of type RULE_02 to the DP60ViewModel.
- `MenuButton menu_rule_2`: sends an EventObject of type RULE_02_OVER to the DP60ViewModel.
- `MenuButton menu_rule_3`: sends an EventObject of type RULE_03 to the DP60ViewModel.

The Nodes button_rule_1 and menu_rule_1 have their visibility properties bound to the BooleanProperty rule_1_visible in the DP60ViewModel.
The Nodes button_rule_2 and menu_rule_2 have their visibility properties bound to the BooleanProperty rule_2_visible and menu_rule_3 is bound to rule_3_visible.

Another difference is the EventHandler for the ObservableList<Line> lines. The EventHandler distinguishes between an element was added or was removed.
If an element was removed, then it clears all the content in the center node and the right node. On the other hand if an element was added then the Hbox_Factory will generate an HBox element for the line, where the HBox already contains a VBox container with the calculation. This VBox was removed from the HBox and the EventHandlers were added to the calculations as it was for the Back Dual Resolution (see section 5.3).
Then the Hbox instance and the Vbox instance were added to the center node and the right node respectively.

The last difference is the BooleanProperty _redraw. Its EventHandler starts when the value it encapsulates changes, i.e. when the value changes from `true` to `false` or from `false` to `true`.
Setting the value to `true` while it is already `true` does not start the EventHandler.
So starting the EventHandler tests if the new value is `true` and then it clears all the content in the center node and the right node, then it generates it again. If the new value is `false` nothing changes.

# Chapter 7

# Conclusion and Outlook

The original goal of implementing all four algorithm was not reached, but the writer of this thesis was quite satisfied with the final result. The goal of having a application with a modern and beautiful GUI as well as practical and functional was reached.
The data was displayed in a orderly manner and the additional functionality to "highlight" single steps taken by an algorithm works perfectly.
The color scheme chosen was adequate for a serious lecture but colorful such not to bore the viewer.
The Main Menu of the application reacts to every action with a message.
Also the possibility of using a several of propositional formulas in different input languages was implemented.
For the Resolution Algorithm extending the display of the deduction tree to any clause present in the calculation was realized.

The only procedure that was not implemented was the Davis-Putnam-Logemann-Loveland (DPLL) algorithm due to several reasons, one being the time factor and the other is that the widely known algorithm is used in its recursive form which did not fit the intended purposes of this application.
An iterative method would have been possible by using stacks to keep track of the branching literals. An implementation for the DPLL algorithm could be based on the existing implementation done for the DP60 algorithm by changing and adapting the DP60 class to the iterative version of the DPLL algorithm. Eventually a method to calculate the third rule of the DPLL algorithm might be needed in the DP60_Utility.

# Bibliography

[1] https://mdaines.github.io/grammophone/, on July 28th at 8 am.

[2] https://material.io/, on July 28th at 9 am.

[3] https://amyfowlersblog.wordpress.com/2010/05/26/javafx-1-3-growing-shrinking-and-filling/, on July 28th at 9:30 am.

[4] Mordechai, Ben-Ari, Mathematical Logic for Computer Science, 3rd Edition, Springer Verlag, 2012

[5] https://nirajrules.wordpress.com/2009/07/18/mvc-vs-mvp-vs-mvvm/, on July 28th at 10 am.

[6] http://www.dotnetcurry.com/xamarin/1382/mvvm-in-xamarin-forms, on July 28th at 11 am.

[7] http://cogitolearning.co.uk/2013/03/writing-a-parser-in-java-introduction/, on July 28th at 12 am.

# Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift