Sofia University St. Kliment Ohridski

Faculty of Mathematics and Informatics
Department of Mathematical Logic and Applications

Master Thesis

Satisfiability of Connected Contact Logics

Author:
Anton Dudov
Faculty number: 25691
Logic and Algorithms
(Mathematics)

Supervisor:
Prof. Tinko Tinchev

March 1, 2023

Contents

1 Tableau Method For Classic Propositional Logic

1.1 What Is A tableau?

A tableau method is a formal proof procedure. First, it could be used as a refutation procedure: to show that a formula X is valid we begin with some syntactical expression intended to assert it is not valid. This expression is broken down syntactically, generally splitting things into several cases. This part of a tableau procedure - the tableau expansion stage - can be thought of as a generalization of disjunctive normal form expansion. Generally, it moves from formulas to subformulas. Finally, there are rules for closing cases: impossibility conditions based on syntax. If each case closes, the tableau itself is said to be closed. A closed tableau beginning with an expression asserting that X is not valid is a tableau proof of X.

There is a second way of thinking about the tableau method: as a search procedure for models meeting certain conditions. Each branch of a tableau can be considered to be a partial description of a model. In automated theorem-proving, tableaux can be used to generate counter-examples.

The connection between the two roles for tableau - as a proof procedure and as a model search procedure - is simple. If we use tableau to search for a model in which X is false, and we produce a closed tableau, no such model exists, so X must be valid.

Will follow the tableau method described in *Handbook of Tableau Methods* [1] . There are two types of tableaux - signed and unsigned. The signed version is going to be used.

1.2 Classical Propositional Tableau

Let's look into the signed tableau system for classical propositional logic.

First, we need syntactical machinery for asserting the non-validity of a formula, and for doing case analysis. For this purpose two signs are introduced: $\mathbb{T}$ and $\mathbb{F}$, where these are simply two new symbols, not part of the language of formulas. *Signed formulas* are expressions of the form $\mathbb{F}X$ and $\mathbb{T}X$, where X is a formula. The intuitive meaning of $\mathbb{F}X$ is that X is *false* (in some model). Similarly, $\mathbb{T}X$ intuitively asserts that X is *true*. Then $\mathbb{F}X$ is the syntactical device for (informally) asserting the non-validity of X. A tableau proof of X begins with $\mathbb{F}X$.

Next, we need machinery (rules) for breaking signed formulas down and doing a case division. We will define rules for each logical operator ($\neg \wedge \vee \Rightarrow \Leftrightarrow$).

The treatment of **negation** is straightforward: from $\mathbb{T}\neg X$ we get $\mathbb{F}X$ and from $\mathbb{F}\neg X$ we get $\mathbb{T}X$. These rules can be conveniently presented as follows.

$$\frac{\mathbb{T}\neg X}{\mathbb{F}X} \qquad\qquad \frac{\mathbb{F}\neg X}{\mathbb{T}X}$$

The rules for **conjunction** are somewhat more complex. From truth tables we know that if $X \wedge Y$ is *true*, X must be *true* and Y must be *true*. Likewise,

if $X \wedge Y$ is *false*, either X is *false* or Y is *false*. This involves a split into two cases. Corresponding syntactic rules are as follows.

$$\frac{\mathbb{T}X \wedge Y}{\begin{array}{c}\mathbb{T}X\\ \mathbb{T}Y\end{array}} \qquad\qquad \frac{\mathbb{F}X \wedge Y}{\mathbb{F}X \mid \mathbb{F}Y}$$

The rules for **disjunction** are similar. From truth tables we know that if $X \vee Y$ is *true*, either X is *true* or Y is *true*. This involves a split into two cases. Likewise, if $X \vee Y$ is *false*, X must be *false* and Y must be *false*. Corresponding syntactic rules are as follows.

$$\frac{\mathbb{T}X \vee Y}{\mathbb{T}X \mid \mathbb{T}Y} \qquad\qquad \frac{\mathbb{F}X \vee Y}{\begin{array}{c}\mathbb{F}X\\ \mathbb{F}Y\end{array}}$$

The rules for **implication**. From truth tables we know that if $X \Rightarrow Y$ is *true*, either X is *false* or Y is *true*. Likewise, if $X \Rightarrow Y$ is *false*, X must be *true* and Y must be *false*. Corresponding syntactic rules are as follows.

$$\frac{\mathbb{T}X \Rightarrow Y}{\mathbb{F}X \mid \mathbb{T}Y} \qquad\qquad \frac{\mathbb{F}X \Rightarrow Y}{\begin{array}{c}\mathbb{T}X\\ \mathbb{F}Y\end{array}}$$

The rules for **equivalence**. From truth tables we know that if $X \Leftrightarrow Y$ is *true*, either X is *true* and Y is *true* or X is *false* and Y is *false*. Likewise, if $X \Rightarrow Y$ is *false*, either X is *true* and Y is *false* or X is *false* and Y is *true*. Corresponding syntactic rules are as follows.

$$\frac{\mathbb{T}X \Leftrightarrow Y}{\begin{array}{c|c}\mathbb{T}X & \mathbb{F}X\\ \mathbb{T}Y & \mathbb{F}Y\end{array}} \qquad\qquad \frac{\mathbb{F}X \Leftrightarrow Y}{\begin{array}{c|c}\mathbb{T}X & \mathbb{F}X\\ \mathbb{F}Y & \mathbb{T}Y\end{array}}$$

The standard way of displaying tableau is as downward branching trees with signed formulas as node labels. Indeed, the tableau method is often referred to as the tree method. Think of a tree as representing the disjunction of its branches, and a branch as representing the conjunction of the signed formulas on it.

When using a tree display, a tableau expansion is thought of temporally, and one talks about the stages of constructing a tableau, meaning the stages of growing a tree. The rules given above are thought of as branch-lengthening rules. Thus, a branch containing $\mathbb{T}\neg\ X$ can be lengthened by adding a new node to its end, with $\mathbb{F}\ X$ as a label. Likewise, a branch containing $\mathbb{F}\ X \vee\ Y$ can be lengthened with two new nodes, labelled $\mathbb{F}\ X$ and $\mathbb{F}\ Y$ (take the node with $\mathbb{F}\ Y$ as the child of the one labeled $\mathbb{F}\ X$). A branch containing $\mathbb{T}\ X \vee\ Y$ can be split - its leaf is given a new left and a new right child, with one labeled $\mathbb{T}\ X$, the other $\mathbb{T}\ Y$. This is how the schematic rules above are applied to trees.

An important point to note is that the tableau rules are non-deterministic. They say what can be done, not what must be done. At each stage, we choose a signed formula occurrence on a branch and apply a rule to it. Since the order of choice is arbitrary, there can be many tableaux for a single signed formula.

Definition 1.1. *A branch is called* ***closed*** *if it contains a contradiction, For example, if it contains* $\mathbb{T}X$ *and* $\mathbb{F}X$ *for some formula* X*.*

Definition 1.2. *A branch is called* ***finished*** *if all formulas in it are applied, i.e. contains only signed variables(atomic formulas).*

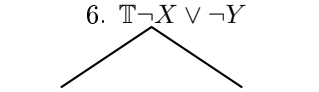
Definition 1.3. *A branch is called* ***opened*** *if it is finished and is not closed.*

Definition 1.4. *A tableau is called* ***closed*** *if all branches are closed.*

Lemma 1.5. *A closed tableau for* $\mathbb{F}X$ *is a tableau proof of* X*, meaning that* X *is a* ***tautology****.*

Here is the tableau expansion beginning with the signed formula $\mathbb{F}(X \wedge Y) \Rightarrow (\neg X \wedge \neg Y)$.

1. $\mathbb{F}(X \wedge Y) \Rightarrow \neg(\neg X \vee \neg Y)$
2. $\mathbb{T}X \wedge Y$
3. $\mathbb{F}\neg(\neg X \vee \neg Y)$
4. $\mathbb{T}X$
5. $\mathbb{T}Y$
6. $\mathbb{T}\neg X \vee \neg Y$



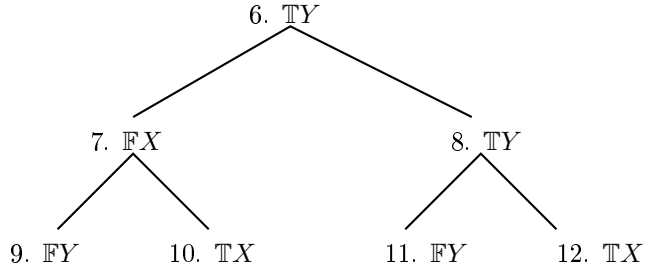
7. $\mathbb{T}\neg X$ 8. $\mathbb{T}\neg Y$
9. $\mathbb{F}X$ 10. $\mathbb{F}Y$

The numbers are for reference purposes. Items 2 and 3 are from 1 by $\mathbb{F} \Rightarrow$. 4 and 5 are from 2 by $\mathbb{T}\wedge$. 6 is from 3 by $\mathbb{F}\neg$. 7 and 8 are from 6 by $\mathbb{T}\vee$. 9 is from 7 by $\mathbb{T}\neg$. 10 is from 8 by $\mathbb{T}\neg$.

The tableau displayed above is closed, so the formula $(X \wedge Y) \Rightarrow (\neg X \wedge \neg Y)$ is a tautology.

It may happen that no tableau proof is forthcoming, and we can think of the tableau construction as proving with counterexamples. Consider the following attempt to prove $(X \Rightarrow Y) \Rightarrow ((Y \Rightarrow X) \Rightarrow \neg Y)$

1. $\mathbb{F}((X \Rightarrow Y) \Rightarrow ((Y \Rightarrow X) \Rightarrow \neg Y))$
2. $\mathbb{T}X \Rightarrow Y$
3. $\mathbb{F}((Y \Rightarrow X) \Rightarrow \neg Y)$
4. $\mathbb{T}Y \Rightarrow X$
5. $\mathbb{F}\neg Y$
6. $\mathbb{T}Y$



7. $\mathbb{F}X$ 8. $\mathbb{T}Y$
9. $\mathbb{F}Y$ 10. $\mathbb{T}X$ 11. $\mathbb{F}Y$ 12. $\mathbb{T}X$

Item 2 and 3 are from 1 by $\mathbb{F} \Rightarrow$, as are 4 and 5 from 3. Item 6 is from 5 by $\mathbb{F}\neg$. Items 7 and 8 are from 2 by $\mathbb{T} \Rightarrow$, as are 9 and 10 from 4. Items

11 and 12 are also from 4 by $\mathbb{T} \Rightarrow$. The leftmost branch is closed because of 6 and 9. The left-right branch is closed because of 7 and 10. The right-left branch is closed because of 8 and 11. But the rightmost branch is not closed. Notice that every non-atomic signed formula has had a rule applied to it on this branch and there is nothing left to do. (For classical propositional logic it is sufficient to apply a rule to a formula on a branch only once.) In fact the branch yields a counterexample, as follows. Let v be a propositional valuation that maps X to *true* and Y to *true* in accordance to 8 and 12. Now, we work our way back up the branch. Since $v(Y) = true$, $v(\neg Y) = false$, item 5. From $v(X) = true$ follows that $v(Y \Rightarrow X) = true$, item 4. From $v(Y) = true$ follows that $v(X \Rightarrow Y) = true$, item 2. Since $v(Y \Rightarrow X) = true$ and $v(\neg Y) = false$ we have $v((Y \Rightarrow X) \Rightarrow \neg Y) = false$, item 3. Finally, $v((X \Rightarrow Y) \Rightarrow ((Y \Rightarrow X) \Rightarrow \neg Y)) = false$, item 1.

Later, we are going to use the tableau method to produce a model in which the initial formula is valid.

From a different point of view, we can think of a classical tableau simply as a set of sets of signed formulas: a tableau is the set of its all branches, and a branch is the set of signed formulas that occur on it. Semantically, we think of the outer set as the disjunction of its members, and these members, the inner sets, as conjunctions of signed formulas they contain. Considered this way, a tableau is a generalization of the disjunctive normal form (a generalization because formulas more complex than literals can occur). Now, the tableau construction process can be thought of as a variation of the process of converting a formula into a disjunctive normal form.

2 Contact Logic

2.1 Syntax

The language of contact logic consist of:

- *Boolean variables* (a denumerable set $\mathcal{V}$)
- *Boolean constants*: 0 and 1
- *Boolean operations*:
  - $\sqcap$ boolean meet
  - $\sqcup$ boolean join
  - * boolean complement
- *Boolean terms* (or simply *terms*)
- *Propositional connectives*: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- *Propositional constants*: $\top$ and $\bot$
- *Modal connectives*: $\leq$(part-of) and C(contact)
- *Complex formulas* (or simply *formulas*)

Terms are defined by the following inductive definition:

- Each Boolean variable is a term
- Each Boolean constant is a term
- If a is a term then a^* is a term
- If a and b are terms then $a \sqcap b$ and $a \sqcup b$ are terms

Atomic formulas are of the form $a \leq b$ and aCb, where a and b terms.

Formulas are defined by the following inductive definition:

- Each propositional constant is a formula
- Each atomic formula is a formula
- If ϕ is a formula then $\neg\phi$ is a formula
- If ϕ and ψ are formulas then $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$ and $(\phi \Leftrightarrow \psi)$ are formulas

Abbreviations: $a = b \stackrel{\text{def}}{=\!=} (a \leq b) \wedge (b \leq a)$, $a \not\leq b \stackrel{\text{def}}{=\!=} \neg(a \leq b)$, $a \neq b \stackrel{\text{def}}{=\!=} \neg(a = b)$

2.2 Relational semantics

Let $\mathcal{F}$ = (W, R) be a relational system with W $\neq \emptyset$ and R $\subseteq W^2$. We call such systems *frames*. Following Galton we may give a spatial meaning of frames naming the elements of W, *cells* and the relation R, *adjacency relation*. Then $\mathcal{F}$ is called *adjacency space*. An example of adjacency space is the chessboard table, the cells are the squares, and two squares are adjacent if they have a common point.

Originally Galton assumed R to be a reflexive and symmetric relation but it is more natural for R to be an arbitrary relation. *Regions* in an adjacency space are arbitrary subsets of W.

Definition 2.1. *By a* ***valuation*** *of the Boolean variables in $\mathcal{F}$ we mean any function $\upsilon : \mathcal{V} \to \mathcal{P}(W)$ assigning to each Boolean variable b a subset $\upsilon(b) \subseteq W$. The valuation υ is then extended inductively to all Boolean terms as follows:*

- $\upsilon(0) = \emptyset$
- $\upsilon(1) = W$
- $\upsilon(a \sqcap b) = \upsilon(a) \cap \upsilon(b)$
- $\upsilon(a \sqcup b) = \upsilon(a) \cup \upsilon(b)$
- $\upsilon(a^*) = W \setminus \upsilon(a)$

Definition 2.2. *(**Contact**) Let a and b are terms. Then*

$$aCb \iff (\exists x \in \upsilon(a))(\exists y \in \upsilon(b))(xRy) \tag{1}$$

Definition 2.3. *The pair $\mathcal{M} = (\mathcal{F}, \upsilon)$ is called* ***model****. The truth of a formula ϕ in $\mathcal{M}$ ($\mathcal{M} \models \phi$ or $\mathcal{F}, \upsilon \models \phi$) is extended inductively to all Boolean terms as follows:*

- *For atomic formulas:*
  - $\mathcal{M} \models \top$
  - $\mathcal{M} \not\models \bot$
  - $\mathcal{M} \models a \leq b \iff \upsilon(a) \subseteq \upsilon(b)$
  - $\mathcal{M} \models aCb \iff (\exists x \in \upsilon(a))(\exists y \in \upsilon(b))(xRy)$
- *For complex formulas:*
  - $\mathcal{M} \models \neg\phi \iff \mathcal{M} \not\models \phi$
  - $\mathcal{M} \models \phi \wedge \psi \iff \mathcal{M} \models \phi$ *and* $\mathcal{M} \models \psi$
  - $\mathcal{M} \models \phi \vee \psi \iff \mathcal{M} \models \phi$ *or* $\mathcal{M} \models \psi$
  - $\mathcal{M} \models \phi \Rightarrow \psi \iff \mathcal{M} \not\models \phi$ *or* $\mathcal{M} \models \psi$
  - $\mathcal{M} \models \phi \Leftrightarrow \psi \iff (\mathcal{M} \models \phi$ *and* $\mathcal{M} \models \psi)$ *or* $(\mathcal{M} \not\models \phi$ *and* $\mathcal{M} \not\models \psi)$

Let us note that in the above semantics we evaluate formulas not locally at points, as it is in the standard modal semantics, but globally in the whole model and this is one of the main differences of the present modal approach with the standard Kripke approach.

Definition 2.4. *A model $\mathcal{M}$ is a* ***model of a formula*** *ϕ if ϕ is true in $\mathcal{M}$.*

Definition 2.5. *If ϕ has a model $\mathcal{M}$, then ϕ is* ***satisfiable****.*

Definition 2.6. *$\mathcal{M}$ is a* ***model of a set of formulas*** *A if $\mathcal{M}$ is a model of all formulas from A.*

Definition 2.7. *A formula ϕ* ***is true*** *(or* ***valid****) in a frame $\mathcal{F}$ ($\mathcal{F} \models \phi$), if $\mathcal{M} \models \phi$ for all models $\mathcal{M}$ based on $\mathcal{F}$, i.e. for all valuations υ we have $\mathcal{F},\upsilon \models \phi$.*

Lemma 2.8. *(Equality of terms) Let a and b are terms. Then*
$a = b \implies \upsilon(a) = \upsilon(b)$.

Lemma 2.9. *(Zero term) Let a and b are terms. Then*
$a \leq b \implies a \sqcap b^* = 0$

Lemma 2.10. *(Non-zero term) Let a and b are terms. Then*
$\neg(a \leq b) \implies a \sqcap b^* \neq 0$

Axiom 2.11. *(Reflexivity) Let a be a term. Then*
$a \neq 0 \implies aCa$.

Axiom 2.12. *(Symmetry) Let a and b are terms. Then*
$aCb \iff bCa$.

Lemma 2.13. *(Monotonicity) Let a and b are terms. Then*
$aCb \wedge a \leq a' \wedge b \leq b' \implies a'Cb'$.

Lemma 2.14. *(Distributivity) Let a and b are terms. Then*
$aC(b \sqcup c) \iff aCb \vee aCc, \quad (a \sqcup b)Cc \iff aCc \vee bCc$.

Lemma 2.15. *Let a, b, c are terms and f, g are formulas. The following formulas are true:*

- $f \wedge T \implies f$, $T \wedge f \implies f$
- $f \wedge F \implies F$, $F \wedge f \implies F$
- $f \vee T \implies T$, $T \vee f \implies T$
- $f \vee F \implies f$, $F \vee f \implies f$
- $a \sqcap 0 = 0$, $0 \sqcap a = 0$
- $a \sqcup 0 = a$, $0 \sqcup a = a$
- $a \sqcap 1 = a$, $1 \sqcap a = a$

- $a \sqcup 1 = 1,\ 1 \sqcup a = 1$
- $(a \sqcup b)Cc \iff aCc \vee bCc$
- $(a \sqcup b) \leq c \iff a \leq c \wedge b \leq c$
- $aCb \implies a \neq 0 \wedge b \neq 0$
- $a \sqcap b \neq 0 \implies aCb$
- $a = 0 \vee b = 0 \implies \neg(aCb)$
- $0 \leq a \implies T$
- $a \leq 1 \implies T$
- $0C0 \implies F$
- $aC0 \implies F$
- $1C1 \implies T$
- $aC1 \implies a \neq 0$
- $a \neq 0 \implies aCa$

2.3 Formula satisfiability

Let ψ be a propositional formula. Let us build a tableau beginning with ψ. If the tableau has an opened branch then ψ is satisfiable. Unfortunately, for the contact logic this is not enough because we need to verify the modal connectives ($\leq$ and C).

Let ϕ be a formula. Let us build a tableau beginning with ϕ. Let the tableau has an opened branch B. The branch $\mathbb{B}$ is a set of signed atomic formulas of the following type:

- $\mathbb{T}C(a, b)$
- $\mathbb{F}C(e, f)$
- $\mathbb{T}a \leq b$
- $\mathbb{F}a \leq b$

where a and b are terms.
B is an opened branch, so there are no contradicting formulas in it, i.e. $(\neg\exists X)(\mathbb{T}X \in B \wedge \mathbb{F}X \in B)$. The signed atomic formulas could be written as follows:

- $\mathbb{T}C(a, b) \rightarrow C(a, b)$ (contact)
- $\mathbb{F}C(a, b) \rightarrow \neg C(a, b)$ (non-contact)

- $\mathbb{T}a \leq b \to a \leq b \to a \sqcap b^* = 0 \to g = 0$ (zero term)
- $\mathbb{F}a \leq b \to \neg(a \leq b) \to a \sqcap b^* \neq 0 \to d \neq 0$ (non-zero term)

where a,b,d and g are terms.

All atomic formulas in the branch should be satisfied, so we can think of it as a conjunction of them. Let's call it a **branch conjunction**. It is sufficient to build a satisfiable model for the branch conjunction to verify that ϕ is satisfiable. Building such a model could be done a lot more effective than building a model for an arbitrary formula because it is just a conjunction.

Definition 2.16. *Let ϕ be a formula. Let $\mathcal{T}$ be a tableau beginning with ϕ. Let $\mathbb{B}$ be a set of all atomic signed formulas in an open branch of $\mathcal{T}$. A **branch conjunction** β is the following formula:*

$$\bigwedge_{\mathbb{T}C(a,b)\in\mathbb{B}} C(a,b) \ \wedge \bigwedge_{\mathbb{T}d=0\in\mathbb{B}} d = 0 \ \wedge \bigwedge_{\mathbb{F}C(e,f)\in\mathbb{B}} \neg C(e,f) \ \wedge \bigwedge_{\mathbb{F}g=0\in\mathbb{B}} g \neq 0 \quad (2)$$

2.4 Branch conjunction model building

The branch conjunction formula β is satisfiable if β has a model $\mathcal{M}$. We have to construct such a model $\mathcal{M} = (\mathcal{F}, \upsilon) = ((\mathrm{W}, \mathrm{R}), \upsilon)$.

Let $\mathbf{V}$ be the finite set of all boolean variables in β. Let their count be n.

Let $\mathbf{T}$ be the infinite set of all boolean terms with variables of $\mathbf{V}$.

There are four types of atomic formulas in β, namely contacts, non-contacts, zero terms and non-zero terms. Only contacts and non-zero terms require existence of modal points. The valuation υ should assign a set of modal points to each boolean variable $x \in \mathbf{V}$. Such that the contacts and non-zero terms be satisfied.

Step 2.17. *Creating modal points*

Let $t \in \mathbf{T}$. Let p be a modal point. Let us try to extend υ such that $p \in \upsilon(t)$. By the valuation definition $\upsilon(t)$ is a composition of intersections, unions, and compliments of valuations $\upsilon(x)$ for $x \in \mathbf{V}$. Therefore, the modal point p should be added to zero or more variable valuations depending on the boolean operations in the term t. With n boolean variables there are 2^n ways of adding the point p. Note that it might not be possible to adjust the variable valuations such that $p \in \upsilon(t)$. For example, if t is $a \sqcap a^* \neq 0$.

Step 2.18. *Creating modal points for non-zero terms*

Let $g \neq 0 \in \beta$. If it is not possible to create a modal point p such that $p \in \upsilon(t)$, then there is no satisfaible model for β.

Step 2.19. *Creating modal points for contacts*

Let $C(a,b) \in \beta$. If it is not possible to create a modal point p_a such that $p_a \in \upsilon(a)$, then there is no satisfaible model for β. Analogously for the term b.

Definition 2.20. *A **variable evaluation** $\mathcal{E}_n$ for n boolean variables is a sequence of 1s and 0s, as follows:*

$$\mathcal{E}_n = < e_1, e_2, \ldots, e_n >, \text{ where } e_1, \ldots, e_n \in \{0, 1\} \quad (3)$$

Let X be the union of all evaluations of boolean variables in $\mathbf{V}$.

$$X = \bigcup_{x \in \mathbf{V}} \upsilon(x) \tag{4}$$

For n boolean variables there are 2^n unique variable evaluations. We will define the modal points as variable evaluations. By the definition of the valuation, it is not possible to distinguish two or more different modal points in some subsets. For example, the W $\setminus X$ subset. The modal point representation also has this limitation. Therefore, it is sufficient to work with the 2^n unique modal points while building the model.

Definition 2.21. *Let $\mathcal{E}_n$ be a variable evaluation for n boolean variables. Then $(\mathcal{E}_n)^i$ is the i-th element in the sequence $\mathcal{E}_n$.*

Definition 2.22. *The set of **all unique variable evaluations** W_n over n variables is defined as follows:*

$$W_n = \{< e_1, e_2, \ldots, e_n > |\ e_1, \ldots, e_n \in \{0, 1\}\} \tag{5}$$

Definition 2.23. *Let $\mathcal{V}_n$ be a finite set of n boolean variables. Let W_n be the set of all unique points over n variables. Then the **valuation** $\upsilon_n : \mathcal{V}_n \to \mathcal{P}(W_n)$ is defined as follows:*

$$\upsilon_n(x_i) = \{\mathcal{E}_n \mid \mathcal{E}_n \in W_n \text{ and } (\mathcal{E}_n)^i = 1\},\ \text{for } x_i \in \mathcal{V}_n \tag{6}$$

It is then extended inductively to all boolean terms as standard valuation as follows:

- *Boolean constants*
  - $\upsilon_n(0) = \emptyset$
  - $\upsilon_n(1) = W_n$
- *Boolean terms*
  - $\upsilon_n(a \sqcap b) = \upsilon_n(a) \cap \upsilon_n(b)$
  - $\upsilon_n(a \sqcup b) = \upsilon_n(a) \cup \upsilon_n(b)$
  - $\upsilon_n(a^*) = W_n \setminus \upsilon_n(a)$

Lemma 2.24. *The zero terms in β are satisfied if each zero term's valuation is the empty set.*

$$g = 0 \in \beta \to \upsilon_n(g) = \emptyset \tag{7}$$

Lemma 2.25. *The non-contacts in β are satisfied if:*

$$\neg C(e, f) \in \beta \to \neg(\exists x \in \upsilon_n(e))(\exists y \in \upsilon_n(f))(xRy) \tag{8}$$

Definition 2.26. *Let $\mathcal{E}_n \in W_n$. $\mathcal{E}_n$ is a* **valid modal point** *of β if it preserve the satisfiability of the zero terms and non-contacts in β:*

$$g = 0 \in \beta \rightarrow \mathcal{E}_n \notin \upsilon_n(g)$$
$$and$$
$$\neg C(e, f) \in \beta \rightarrow \mathcal{E}_n \notin (\upsilon_n(e) \cap \upsilon_n(f))$$

Definition 2.27. *Let $W^v \subseteq W_n$ be the set of all valid modal points of β.*

$$W^v = \{\mathcal{E}_n \mid \mathcal{E}_n \in W_n \text{ and } \mathcal{E}_n \text{ is a valid modal point of } \beta\} \tag{9}$$

Definition 2.28. *Let x, $y \in W^v$ are valid modal points. Then $\langle x, y\rangle$ is a* **valid connected pair** *of β modal points if it preserves the satisfiability of non-contacts in β.*

$$\neg C(e, f) \in \beta \rightarrow \neg(x \in \upsilon_n(e) \text{ and } y \in \upsilon_n(f)) \text{ or} (x \in \upsilon_n(f) \text{ and } y \in \upsilon_n(e)) \tag{10}$$

Definition 2.29. *Let $\eta : (\mathbf{T} \times W_n) \rightarrow \{0, 1\}$ be a function which defines whether a modal point is in the valuation of a term. Let $t \in \mathbf{T}$. Let $\mathcal{E}_n \in W_n$. The inductive definition of η on the structure of the term t is as follows:*

- $\eta(0, \mathcal{E}_n) = 0$
- $\eta(1, \mathcal{E}_n) = 1$
- $\eta(x_i, \mathcal{E}_n) = 1 \iff (\mathcal{E}_n)^i = 1$
- $\eta(a \sqcap b, \mathcal{E}_n) = 1 \iff \eta(a, \mathcal{E}_n) = 1$ *and* $\eta(b, \mathcal{E}_n) = 1$
- $\eta(a \sqcup b, \mathcal{E}_n) = 1 \iff \eta(a, \mathcal{E}_n) = 1$ *or* $\eta(b, \mathcal{E}_n) = 1$
- $\eta(a^*, \mathcal{E}_n) = 1 \iff \eta(a, \mathcal{E}_n) = 0$

Definition 2.30. *Let $IsValidCon : (W^v \times W^v) \rightarrow \{0, 1\}$ be a function which defines whether a connected pair is valid.*

$$IsValidCon(x, y) = 1 \iff \langle x, y\rangle \text{ is a valid connected pair} \tag{11}$$

Lemma 2.31. *Let $t \in \mathbf{T}$ be an arbitrary term. By the definition of η and υ_n follows:*

$$\eta(t, \mathcal{E}_n) = 1 \iff \mathcal{E}_n \in \upsilon_n(t) \tag{12}$$

The algorithm described bellow creates modal points and connections only for the non-zero and contact terms in β.

```
1  W ← ∅
2  R ← ∅
   /* Process the non-zero terms in β                                   */
3  for d ≠ 0 ∈ β do
4  |  for E_n ∈ W^v do
5  |  |  if η(d, E_n) = 1 then
6  |  |  |  W ← W ∪ {x}
7  |  |  |  R ← R ∪ {⟨x, x⟩}
8  |  |  |  go to 3
9  |  end
10 |  Unable to construct a model.
11 end
   /* Process the contacts in β                                         */
12 for C(a, b) ∈ β do
13 |  for x, y ∈ W^v do
14 |  |  if η(a, x) = 1 ∧ η(b, x) = 1 ∧ IsValidCon(x, y) then
15 |  |  |  W ← W ∪ {x, y}
16 |  |  |  R ← R ∪ {⟨x, x⟩, ⟨y, y⟩, ⟨x, y⟩, ⟨y, x⟩}
17 |  |  |  go to 12
18 |  end
19 |  Unable to construct a model.
20 end
21 Successfully constructed a model M = ((W, R), υ_n).
```

2.5 Formal system

Let's describe a contact logic formal system.

All lemmas are deducible from the syntax and semantics. They are syntactically checked. Therefore, they are syntactically correct.

Let ϕ be a formula. If the model building procedure creates a model for $\neg\phi$, then ϕ is not a tautology. The process of building the model is the proof.

If the model building procedure creates a model for ϕ, then ϕ is satisfiable and the model is the proof. Otherwise, the process of building a model is a proof that ϕ is not satisfiable.

3 Connected Contact Logic

3.1 Connectivity

In topology and related branches of mathematics, a connected space is a topological space that cannot be represented as the union of two disjoint non-empty open subsets. Connectedness is one of the principal topological properties that are used to distinguish topological spaces.

Axiom 3.1. *(Connectivity) Let b be a term. Then*

$$b \neq 0 \wedge b \neq 1 \implies bCb^* \tag{13}$$

Let $\mathcal{F} = (W, R)$ be a relational system with W $\neq \emptyset$, R $\subseteq W^2$ and a, b are terms. Let us recall the definition of C:

$$aCb \iff (\exists x \in \upsilon(a))(\exists y \in \upsilon(b))(xRy)$$

The connectivity axiom can be written as follows:

$$\upsilon(b) \neq \emptyset \wedge \upsilon(b) \neq W \implies (\exists x \in \upsilon(b))(\exists y \in W \setminus \upsilon(b))(xRy) \tag{14}$$

The relational system $\mathcal{F} = (W, R)$ defines an unidirected graph G(W, R). W is the set of vertices and R is the set of edges.

Definition 3.2. *Let G = (W, R) be a graph. W is the set of vertices and R the set of edges. A **path** $\pi_G(x, y)$ is a sequence of vertices $(x, v_1, \ldots, v_k, y)$ such that $x, v_1, \ldots, v_k, y \in V$ and $xRv1, v_1Rv_2, \ldots, v_{k-1}Rv_k, v_kRy$.*

Definition 3.3. *Let $G = (W, R)$ be an undirected graph. W is the set of vertices and R the set of edges. G is* **connected** *if there is a path between every two different vertices in W.*

$$x, y \in W \to (x \neq y \implies \pi_G(x, y))$$

Theorem 3.4. *(Connectivity) Let $\mathcal{F} = (W, R)$ be a relational system. Let G = (W, R) be the undirected graph defined by $\mathcal{F}$.*

$$\textit{connectivity axiom is satisfied in } \mathcal{F} \iff G \textit{ is connected} \tag{15}$$

Definition 3.5. *Let $\mathcal{F}$ = (W, R) be a relational system. Let G = (W, R) be the undirected graph defined by $\mathcal{F}$. Let $\mathcal{M} = (\mathcal{F}, \upsilon_n)$ be a model of β. $\mathcal{M}$ is a* ***connected model*** *if $G = (W, R)$ is connected. Thus, the connectivity axiom is satisfied in $\mathcal{F}$.*

3.2 Connected model building

Let β be a branch conjunction as in 2.16 :

$$\bigwedge_{\mathbb{T}C(a,b)\in\mathbb{B}} C(a,b) \ \wedge \bigwedge_{\mathbb{T}d=0\in\mathbb{B}} d=0 \ \wedge \bigwedge_{\mathbb{F}C(e,f)\in\mathbb{B}} \neg C(e,f) \ \wedge \bigwedge_{\mathbb{F}g=0\in\mathbb{B}} g\neq 0$$

Definition 3.6. *Let W^v be the set of all valid modal points of β. Let $R^v \subseteq W^{v2}$ be the set of all valid connected pairs of β modal points.*

$R^v = \{\langle x, y\rangle \mid x, y \in W^v$ *and* $\langle x, y\rangle$ *is a valid connected pair of β modal points*$\}$

Step 3.7. *Let $\mathcal{F}^v = (W^v, R^v)$ be a relational system and $\mathcal{M}^v = (\mathcal{F}^v, \upsilon_n)$ be a model in this system. $\mathcal{M}^v$ contains all valid modal points and valid connections between them w.r.t zero terms and non-contacts in β. $\mathcal{M}^v$ is a model of β if the contacts and non-zero terms in β are satisfied. If $\mathcal{M}^v$ is not a model of β, then β does not have a model. There is no way of adding new modal points or relations to satisfy the contacts and non-zero terms. Therefore, there is no connected model eigther.*

Definition 3.8. *Let $G = (W, R)$ be a graph. Let $G' = (W', R')$. G' is a* ***subgraph*** *of G ($G' \subseteq G$) if:*

$$W' \subseteq W \text{ and } R' = \{\langle x, y\rangle \mid x, y \in W' \text{ and } xRy\}$$

Lemma 3.9. *Let $\mathcal{F} = (W, R)$ be a relational system and $\mathcal{M} = (\mathcal{F}, \upsilon_n)$ be a model. Let $G = (W, R)$ be the graph of $\mathcal{F}$. Let $G' = (W', R') \subseteq G$. Then G' defines a model $\mathcal{M}' = ((W', R'), \upsilon'_n)$. Where υ'_n is defined by the following inductive definition:*

- $\upsilon'_n(x_i) = \upsilon_n(x_i) \cap W'$ *for each boolean variable* x_i
- $\upsilon'_n(0) = \emptyset$
- $\upsilon'_n(1) = W'$
- $\upsilon'_n(a \sqcap b) = \upsilon'_n(a) \cap \upsilon'_n(b)$
- $\upsilon'_n(a \sqcup b) = \upsilon'_n(a) \cup \upsilon'_n(b)$
- $\upsilon'_n(a^*) = W' \setminus \upsilon'_n(a)$

Lemma 3.10. *Let $\mathcal{M}^v = (\mathcal{F}^v, \upsilon_n)$ be a model containing all valid modal points and relations. Let $G^v = (W^v, R^v)$ be the graph of $\mathcal{F}^v$. Let $G = (W, R) \subseteq G^v$ and $\mathcal{M} = ((W, R), \upsilon'_n)$ be the model defined from G by lemma 3.9. Then:*

- *M preserves the satisfiability of the contacts in β if*

$$C(a,b) \in \beta \rightarrow (\exists x \in \upsilon'_n(a))(\exists y \in \upsilon'_n(b))(xRy)$$

- *M preserves the satisfiability of the non-contacts in β if*

$$\neg C(e,f) \in \beta \rightarrow \neg((\exists x \in \upsilon'_n(e))(\exists y \in \upsilon'_n(f))(xRy))$$

- *M preserves the satisfiability of the zero terms in β if*

$$d = 0 \in \beta \rightarrow \upsilon'_n(d) = \emptyset$$

- *M preserves the satisfiability of the non-zero terms in β if*

$$g \neq 0 \in \beta \rightarrow \upsilon'_n(g) \neq \emptyset$$

Definition 3.11. *Let $G = (W, R)$ be a graph. Let $G' = (W', R') \subseteq G(W, R)$. If G' is connected, then G' is a* ***connected component*** *of G.*

Definition 3.12. *Let $G = (W, R)$ be a graph. Let $G' = (W', R')$ be a connected component of G. G' is a* ***maximal connected component*** *of G if:*

$$x \in W' \rightarrow \neg(\exists y \in W \setminus W')(xRy)$$
$$x, y \in W' \rightarrow xRy \iff xR'y$$

Step 3.13. *Let $\mathcal{M}^v$ be a model of β. Let $G^v = (W^v, R^v)$ be the graph of $\mathcal{F}^v$. All models defined by the connected components of G^v preserve the satisfiability of the zero terms and non-contacts in β. If there is one which satisfies the contacts and non-zero terms in β, then this is a connected model of β.*

Let n be the number of elements in W^v. There are 2^n subgraphs of G^v. It is slow to check each one of them if it's connected and whether it satisfies the contacts and non-zero terms of β. It is sufficient to split G^v to it's maximal connected components and check only them.

Definition 3.14. *Let $G = (W, R)$ be a graph. Let $Comp^G$ be the set of all maximal connected components of G.*

$$Comp^G = \{G' \mid G' \subseteq G \text{ and } G' \text{ is maximal connected component}\}$$

Definition 3.15. *Let $G' = (W', R')$ and $G'' = (W'', R'')$ are graphs. The union of these graphs is the graph $G = (W, R) = (W' \cup W'', R' \cup R'')$.*

Lemma 3.16. *Let $G = (W, R)$ be a graph. Let $Comp^G$ be the set of all maximal connected components of G. The union of $Comp^G$ is the graph G.*

$$G = \bigcup_{G' \in Comp^g} G'$$

Lemma 3.17. *Let $G = (W, R)$ be a graph. Let $Comp^G$ be the set of all maximal connected components of G. Let $G' = (W', R')$, $G'' = (W'', R'') \in Comp^G$. Then G' and G" does not have a common vertices and edges.*

$$W' \cap W'' = \emptyset \text{ and } R' \cap R'' = \emptyset$$

Step 3.18. *Let $Comp^{G^v}$ is the set of the maximal connected components of G^v. Let n be the number of elements in W^v. The number elements in $Comp^{G^v}$ is at most n. The maximal connected components of G^v does not have common vertices nor edges. Therefore, the splitting of G to it's maximal connected components is a fast and simple operation.*

Step 3.19. *Each model defined by a graph in $Comp^{G^v}$ satisfies the zero terms and non-contacts of β. They do not introduce new modal points netiher a connections. If there is one which satisfies the contacts and non-zero terms of β, then this is a connected model of β. Otherwise, β does not have a connected model.*

Lemma 3.20. *(Connected component extension)*

Let $G = (W, R)$ be a graph. Let $G' = (W', R') \subseteq G$ be a connected component of G. G' could be extended to a maximal connected component $G_m(W_m, R_m)$ of G as follows.

$$W_m = W' \cup \{x \mid x \in W \setminus W' \text{ and } (\exists y \in W')(\pi_G(x, y))\}$$
$$R_m = \{\langle x, y\rangle \mid \langle x, y\rangle \in R \text{ and } x, y \in W_m\}$$

Theorem 3.21. *Let $G^v = (W^v, R^v)$ be the graph of $\mathcal{F}^v$. If G^v does not have a maximal connected component which defines a model of β, then β does not have a connected model.*

Proof. Let G^v does not have a maximal connected component which defines a model of β. Let β have a connected model $\mathcal{M} = (\mathcal{F}(W, R), \upsilon''_n)$. Let $G = (W, R)$ be the graph defined by $\mathcal{F}$. From the definition of the connected model follows that G is connected. G contains only valid modal points and valid relations between them. G^v contains all valid modal points and all valid connections between them. Therfore, $G \subseteq G^v$ and G is a connected component of G^v. By lemma 3.20 G can be extended to a maximal connected component $G_m = (W_m, R_m)$ of G^v. Let $\mathcal{M}_m = (\mathcal{F}_m(W_m, R_m), \upsilon'_n)$ be the model defined from G_m by lemma 3.9. The extension adds points from W^v and relations from R^v to G_m. These are only valid modal points and valid connections between them. Therefore, M_m preserves the satisfiability of the non-contacts and zero terms in β. G_m keeps the points and relations from G. Thus, M_m preserves the satisfiability of the contacts and non-zero terms in β. Hence, M_m is a connected model of β. This leads to a contradiction with the assumption that G^v does not have a maximal connected component which defines a model of β. □

4 Implementation Introduction

The main programming language is C++. Flex & Bison [4] [5] are used to parse the input formula. The formula proover is a C++ library. The unit and performance tests are C++ applications. The user application is a Web page. The Web server is implemented with the third-party CppRestSDK [6] library. The satisfiability checking runs on the server. There is a feature to interrupt an ongoing process. Can be triggered by the user via a button. There is a user disconnecting detection which cancels the requested formula prooving.

The project repository is at `https://github.com/Anton94/modal_logic_formula_prover`. Each commit is build and tested on a various compilers (Windows and Linux OS).

4.1 Syntax

The formula should be easy and intuitive to write. Only the keyboard keys should be used. The legend bellow describes the formula's syntax:

Terms		
0	0	Boolean constant 0
1	1	Boolean constant 1
-	*	Boolean complement
*	$\sqcap$	Boolean meet
+	$\sqcup$	Boolean join
()	()	Parentheses
[a-zA-Z0-9]+	x_1	Boolean variable. Syntax $x1$, $Y42$, $Var101$
Formulas		
F	$\bot$	Propositional constant false
T	$\top$	Propositional constant true
~	$\neg$	Negation
&	$\wedge$	Conjunction
|	$\vee$	Disjunction
->	$\Rightarrow$	Implication
<->	$\Leftrightarrow$	Equivalence
C	C	Contact, syntax C(t_1, t_2)
<=	$\leq$	Part of , syntax <=(t_1, t_2)
<=m	$\leq m$	Measured Part of, syntax <=m(t_1, t_2)
=0	=0	Zero term, syntax $t_1 = 0$
()	()	Parentheses

These are a few examples of formulas:

- $C(x1 * 1, x2 + y1)$
- $C(x1 + 0, (-x2 + x3) * x1)$
- $C(x1, x2)$ & $C(x2, x3)$ & $\sim C(x1, x3)$

- $C(x1, x2)$ & $<= (x1, x3)$ & $\sim C(x2, x3)$
- $C(x1, x2)$ -> $C(x2, x1)$
- $C(x1, x2)$ & $C(x2, x3)$ => $C(x1, x3)$
- F -> $C(x1, x2)$ & $\sim C(x1, x2)$

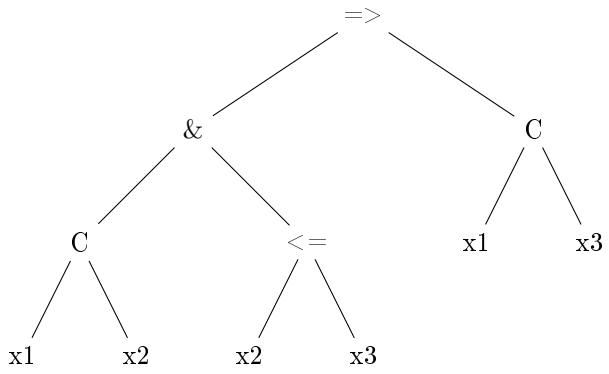
4.2 Formula parsing

The formula is a sequence of characters. These characters does not give us any information for the formula's structure. It should be analyzed. Flex [7] and Bison [8] are used to parse it into an AST (Abstract Syntax Tree) [9]. Flex is used as a tokenizer. Bison is used as the parser.

4.2.1 Abstract Syntax Tree

The AST is a binary tree. Each node has an operation type and up to two children. The formula nodes are prior term nodes. A formula node could have term node as children. Term nodes could not have a formula node as a child. The leaves are variables or constants.

Let $\phi = (C(x1, x2)$ & $<= (x2, x3)) => C(x1, x3)$. The following is an AST of ϕ:



Operation types Enum structure is used to represent the type of formulas and terms in a memory efficient way.

```
ast.h

enum class formula_operation_t
{
    constant_true,
    constant_false,
    conjunction,
    disjunction,
    negation,
    implication,
    equality,
    contact,
    less_eq,
    measured_less_eq,
    eq_zero
};

enum class term_operation_t
{
    constant_true,
    constant_false,
    union_, // union is a keyword
    intersection,
    complement,
    variable
};
```

Node types There are two types of nodes. Formula nodes and term nodes. They are defined with a separate classes.

```
ast.h

class Node
{
    ...
};

class NFormula : public Node
{
public:
    NFormula(formula_operation_t op,
             Node* left = nullptr, Node* right = nullptr);
    ...

    formula_operation_t op;
    Node* left;
    Node* right;
};

class NTerm : public Node
{
public:
    NTerm(term_operation_t op,
          NTerm* left = nullptr, NTerm* right = nullptr);
    ...

    term_operation_t op;
    NTerm* left;
    NTerm* right;
    std::string variable;
};
```

4.2.2 Tokenizer

The tokenizer is responsible for demarcating the special sybmols in the input formula. After the symbols are identified a token is created for each of them or at least for those significant to the semantic of the input formula. For example, the whitespaces are not significant and a tokens are not created for them. We shall use Flex as a tokenizer [7].

Grammar The tokenizer's grammar is composed from two types of tokens. Single character and multi character.

The Single character tokens are directly matched in the input formula and are representing the token itself. The multi character token is a sequence of characters which have some meaning when bundled together. This tokenizer's grammar is unambiguous and each input formula is uniquely tokenized.

The tokens derivation is explaned in details in the following table with Flex syntax. The matched symbol represents the symbol from the input formula and the output token is the newly created token for the matched symbol.

Matched sequence	Output token
[\t\n]	;
[,TF01()C&| *+-]	yytext[0];
"<="	T_LESS_EQ;
"<=m"	T_MEASURED_LESS_EQ;
"= 0"	T_EQ_ZERO;
"->"	T_FORMULA_OP_IMPLICATION;
"<->"	T_FORMULA_OP_EQUALITY;
[a-zA-Z0-9]+	T_STRING;
.	yytext[0];

Let us review the table above. All white spaces, tabulations and newlines are ignored. The syntax for it is the **;** character.

All single character tokens are passed as their ASCII code. The syntax for it is **yytext[0]**. It gives the matched character. That way it will be easy to use them in the parser.

The multi character tokens are converted to unique identificators. For example, the "<=" sequence is converted to T_LESS_EQ. The sequence of letters and numbers is converted to T_STRING. Later, it will be used as a term variable.

The last matched symbol in the table represents everything else, if nothing has been matched then just return the text itself. The parser will use it to promt where the unrecongized symbol was found and the symbol itself can be printed out.

4.2.3 Parser

The single character tokens are passed as their ASCII symbol to Bison. As discussed above the multi character tokens need more clearance in order to represent the literal from the input text symbols. The followings are definition of literals for multi character tokens:

- %token <const char*> T_STRING is the literal for "string"
- %token T_LESS_EQ is the literal for "<="
- %token T_MEASURED_LESS_EQ is the literal for "<=m"
- %token T_EQ_ZERO is the literal for "=0"
- %token T_FORMULA_OP_IMPLICATION is the literal for "->"
- %token T_FORMULA_OP_EQUALITY is the literal for "<->"

The followings are definitions of priority and associativity of the operation tokens. The priority is from low to hight (w.r.t. the line order in which they are defined)

- %left T_FORMULA_OP_IMPLICATION T_FORMULA_OP_EQUALITY

- %left '|' '+'
- %left '&' '*'
- %right '~' '-'
- %nonassoc '(' ')'

Grammar With the usage of the Parser literals, the input formula can be parsed to an Abstract Syntax Tree(AST). The AST contains all the data from the input string formula in a more structed way. On the AST additional optimizations can be done which will simplify the initial formula. It will produce better performance when a model is seeked in the satisfiability algorithms.

For convenience, will define two helper methods. Namely, **create_term_node** and **create_formula_node**. Both method construct AST nodes.

The create_term_node method creates an AST term node. It's arguments are an operation and up to two child terms. Depending on the operation arity.

The create_formula_node is analogious to the create_term_node method. Creates an AST formula node.

Few special symbols to define beforehand:

- **$$** is the return value to the 'parent'. Later, he can use it, e.g. as a child.
- **$i** is the return value of the i-th matched element in the matcher sequence.

Algorithm The following is the parser algorithm which produces an Abstract Sytax Tree.

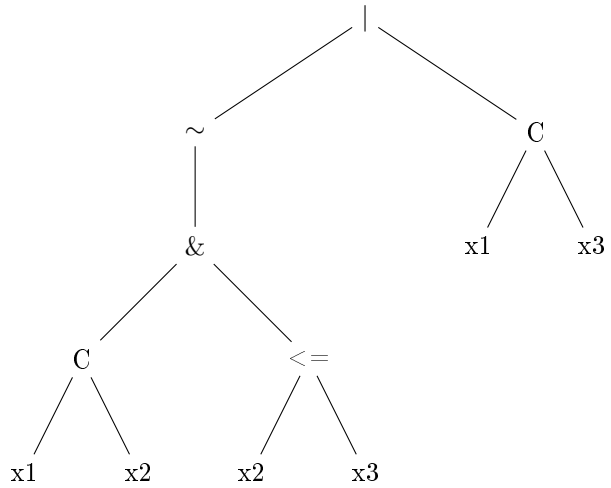
```
parser.y

formula // 'formula' non-terminal
    : 'T' { // matching token 'T'
        $$ = create_formula_node(constant_true);
    }
    | 'F' {
        $$ = create_formula_node(constant_false);
    }
    | 'C' '(' term ',' term ')' {
        $$ = create_formula_node(contact, $3, $5);
    }
    | "<=" '(' term ',' term ')' {
        $$ = create_formula_node(less_eq, $3, $5);
    }
    | "<=m" '(' term ',' term ')' {
        $$ = create_formula_node(measured_less_eq, $3, $5);
    }
    | term "=0" {
        $$ = create_formula_node(eq_zero, $1);
    }
    | '(' formula '&' formula ')' {
        $$ = create_formula_node(conjunction, $2, $4);
    }
    | formula '&' formula {
```

```
        $$ = create_formula_node(conjunction, $1, $3);
    }
    | '(' formula '|' formula ')' {
        $$ = create_formula_node(disjunction, $2, $4);
    }
    | formula '|' formula {
        $$ = create_formula_node(disjunction, $1, $3);
    }
    | '~' formula {
        $$ = create_formula_node(negation, $2);
    }
    | '(' formula "->" formula ')' {
        $$ = create_formula_node(implication, $2, $4);
    }
    | formula "->" formula {
        $$ = create_formula_node(implication, $1, $3);
    }
    | '(' formula "<->" formula ')' {
        $$ = create_formula_node(equality, $2, $4);
    }
    | formula "<->" formula {
        $$ = create_formula_node(equality, $1, $3);
    }
    | '(' formula ')' {
        $$ = $2;
    }
  ;
term
    : '1' {
        $$ = create_term_node(constant_true);
    }
    | '0' {
        $$ = create_term_node(constant_false);
    }
    | "string" {
        $$ = create_term_node(term_operation_t::variable);
        $$->variable = std::move(*$1);
        // the string is allocated from the
        // tokenizer, and we need to free it
        free_lexer_string($1);
    }
    | '(' term '*' term ')' {
        $$ = create_term_node(intersection, $2, $4);
    }
    | term '*' term {
        $$ = create_term_node(intersection, $1, $3);
    }
    | '(' term '+' term ')' {
        $$ = create_term_node(union_, $2, $4);
    }
    | term '+' term {
        $$ = create_term_node(union_, $1, $3);
    }
    | '-' term {
        $$ = create_term_node(complement, $2);
    }
    | '(' term ')' {
```

```
        $$ = $2;
    }
;
```

4.3 Formula refinement

The AST can be easily modified and optimized. One of the modifications is removing the implications and equivalences. They are replaced by conjunction, disjunction, and negation. This is convenient because it simplifies the tableau method. It does not have to handle implication and equivalence operations. The following is a modified AST of ϕ 4.2.1 without the implication:



4.3.1 Visitor Pattern

The AST modification is best achieved with the visitor pattern [10]. Uses double virtual dispatching. Separates the algorithm from the object structure on which it operates. Allows new visitors to be added in a simple manner. Each AST modification will be implemented as a visitor. Will not explain the pattern in depth. In essence, the visitor pattern requires the AST nodes to impelement a virtual **accept** method. This method accepts a visitor as argument and calls the visitor's virtual **visit** method with the real node's type. This is the double virtual dispatching. One virtual call to find the node's real type. Another to find the visitor's real type. Now, adding a new visitor requires only adding it's class. Does not require changes in the AST node classes or other visitor classes.

visitor.h/ast.h

```
...
class Visitor
{
public:
    virtual void visit(NFormula& f) = 0;
    virtual void visit(NTerm& t) = 0;
};

// Example visitor(aglorithm) which will print the AST tree.
class VPrinter : public Visitor
{
public:
    void visit(NFormula& f) override
    {
        // Print the formula node's data.
    }
    void visit(NTerm& t) override
    {
        // Print the term node's data.
    }
};

class Node
{
public:
    virtual void accept(Visitor& v) = 0;
};

class NFormula : public Node
{
public:
    void accept(Visitor& v) override { v.visit(*this); }

    ...
};

class NTerm : public Node
{
public:
    void accept(Visitor& v) override { v.visit(*this); }

    ...
};
```

It's worth showing the implementation of the VPrinter visit methods. They are simple and a good illustration of the pattern. Shows how the AST tree is traversed by calling the VPrinter instance with the AST's root node.

visitor.cpp

```
void VPrinter::visit(NFormula& f)
{
    switch(f.op)
    {
        case formula_operation_t::constant_true:
            out_ << "T"; // out_ is an output stream, e.g. std::cout.
            break;
        case formula_operation_t::constant_false:
            out_ << "F";
            break;
        case formula_operation_t::conjunction:
            out_ << "(";
            f.left->accept(*this);
            out_ << " & ";
            f.right->accept(*this);
            out_ << ")";
            break;
        case formula_operation_t::disjunction:
            // Analogous to conjunction. The symbol is '|'.
        case formula_operation_t::implication:
            // Analogous to conjunction. The symbol is '->'.
        case formula_operation_t::equality:
            // Analogous to conjunction. The symbol is '<->'.
        case formula_operation_t::negation:
            out_ << "~";
            f.left->accept(*this);
            break;
        case formula_operation_t::less_eq:
            out_ << "<=(";
            f.left->accept(*this);
            out_ << ", ";
            f.right->accept(*this);
            out_ << ")";
            break;
        case formula_operation_t::measured_less_eq:
            out_ << "<=m(";
            f.left->accept(*this);
            out_ << ", ";
            f.right->accept(*this);
            out_ << ")";
            break;
        case formula_operation_t::eq_zero:
            out_ << "(";
            f.left->accept(*this);
            out_ << ")=0";
            break;
        case formula_operation_t::contact:
            out_ << "C(";
            f.left->accept(*this);
            out_ << ", ";
            f.right->accept(*this);
```

```
            out_ << ")";
            break;
        default:
            assert(false && "Unrecognized.");
    }
}

void VPrinter::visit(NTerm& t)
{
    switch(t.op)
    {
        case term_operation_t::constant_true:
            out_ << "1";
            break;
        case term_operation_t::constant_false:
            out_ << "0";
            break;
        case term_operation_t::variable:
            out_ << t.variable;
            break;
        case term_operation_t::union_:
            out_ << "(";
            t.left->accept(*this);
            out_ << " + ";
            t.right->accept(*this);
            out_ << ")";
            break;
        case term_operation_t::intersection:
            // Analogous to union_. The symbol is '*'.
        case term_operation_t::complement:
            out_ << "-";
            t.left->accept(*this);
            break;
        default:
            assert(false && "Unrecognized.");
    }
}
```

4.3.2 Visitors Overview

The following are the supported visitors. Their implementation is close to a thousand lines of code and can be checked in the repository.

VReduceConstants Removes all unnecessary children of And/Or/Negation operations of the following type:

• ~T ≡ F	• C(0,0) ≡ F	• ~F ≡ T	• C(1,1) ≡ T
• (T & T) ≡ T	• C(a,0) ≡ F	• (F | F) ≡ F	• C(0,a) ≡ F
• (g & T) ≡ g	• -1 ≡ 0	• (g | T) ≡ T	• -0 ≡ 1
• (T & g) ≡ g	• (1 * 1) ≡ 1	• (T | g) ≡ T	• (0 + 0) ≡ 0
• (g & F) ≡ F	• (t * 1) ≡ t	• (g | F) ≡ g	• (t + 1) ≡ 1
• (F & g) ≡ F	• (1 * t) ≡ t	• (F | g) ≡ g	• (1 + t) ≡ 1
• 0=0 ≡ T	• (t * 0) ≡ 0	• 1=0 ≡ F	• (t + 0) ≡ t
• <=(0,a) ≡ T	• (0 * t) ≡ 0	• <=(a,1) ≡ T	• (0 + t) ≡ t

VConvertContactsWithConstantTerms Converts C with constant 1 terms in !=0 atomic formulas. This visitor is best used after the contacts are reduced, via VReduceConstants

- C(a,1) ≡ ~(a=0)
- C(1,a) ≡ ~(a=0)

VConvertLessEqContactWithEqualTerms Converts C and <= atomic formulas with identical terms:

- <=(a,a) ≡ T,
  since (a * -a = 0))
- C(a,a) ≡ ~(a=0)

VReduceDoubleNegation Removes the double/tripple/etc negations. This visitor is best used after all visitors which might add additional negations!

- -(-g) ≡ g
- -(-t) ≡ t

VConvertImplicationEqualityToConjDisj Converts all formula nodes of type implication and equality to nodes which are using just conjunction and disjunction. The main reason for this visitor is to simplify the formula operations. This visitor simplifies the formula to contain only conjunctions, disjunctions and negation operations.

- (f -> g) ≡ (~f | g)
- (f <-> g) ≡ ((f & g) | (~f & ~g))

VConvertLessEqToEqZero Converts a <= formula to an equals to zero atomic formula

- <=(a,b) ≡ (a * -b) = 0

VSplitDisjInLessEqAndContacts Divides C and <= atomic formulas with a disjunction term into two simpler formulas

- C(a + b, c) ≡ C(a, c) | C(b, c)
- C(a, b + c) ≡ C(a, b) | C(a, c)
- <=(a + b, c) ≡ <=(a,c) & <=(b,c)

There are few visitors which only collect or print information from the formula

- VVariablesGetter - gets all variables from the formula (as string)
- VPrinter - prints the formula to some provided output stream

4.4 Formula building

The formula is a binary tree. A **subformula** is a subtree in the formula's tree. The process of building a satisfiable model is computational heavy. A part of it is lexical comparing and traversing subformulas. The AST is convenient to modify the formula. These modifications are a preprocessing step. After them the formula will not be modifed. A few optimizations could be done.

4.4.1 Optimizations

Reducing formula operations The formula operations could be reduced to not have *implication*, *equivalence* and *less equal*. The *VConvertImplicationEqualityToConjDisj* and *VConvertLessEqToEqZero* visitor should be applied.

Variable substitution The variables are a sequence of characters, i.e. strings. It is slow to compare them. An integer ID could be assigned to each variable. One way to do it is via the *VVariablesGetter* visitor. Retrieve all unique variables in a vector and use their positions as IDs.

Hashing subformulas Conducting a test whether two subformulas are equal in a lexical way is an important procedure for the sake of performance. Such equality checks are required in various situations. One of which is checking if a subformula exists in a set of subformulas.

Have in mind that the naive solution to do an equality check is to compare the whole subformula strcuture. The complexity is O(n), where n is the size of the subformula. To reduce this complexity a precalcaulated hash value shall be used. For each formula node an additional hash variable is stored. The hash is computed recursively through the formula's structure. The hash of a parent node depends on the hashes of its children nodes. The equality comparison first checks the hash codes. The full equality checking is done only for matching hash codes.

4.4.2 Layout

The formula structure will be similar to the AST. A couple of changes reflecting the optimizations above. The following is the final formula node's layout.

formula.h/term.h

```
class formula {
    ...
    enum class operation_type : char {
        constant_true,
        constant_false,
        conjunction,
        disjunction,
        negation,
        measured_less_eq,
        eq_zero,
        c,
        invalid,
    };

    operation_type op_;
    std::size_t hash_;

    struct child_formulas {
        formula* left;
        formula* right;
    };
    struct child_terms {
        term* left;
        term* right;
    };

    union {
        // Holds only one of the described objects.
        // Depending on the operation type the
        // child_f_ or child_t_ is "valid".
        child_formulas child_f_;
        child_terms child_t_;
    };
};

class term {
    ...
    enum class operation_type : char
    {
        constant_true,
        constant_false,
        union_,
        intersection,
        complement,
        variable,
        invalid,
    };

    operation_type op_;
    std::size_t hash_;
```

```
    struct children
    {
        term* left;
        term* right;
    };
    union {
        // Holds either children or variable id.
        // Depending on the node's operation type.
        children children_;
        size_t variable_id_;
    };
};
```

4.4.3 Hashing

The following is the implemented hash construction procedure for the term node. The formula node is analogous.

```
term.cpp

void term::construct_hash()
{
  switch(op_)
  {
  case operation_t::constant_true:
  case operation_t::constant_false:
    break;
  case operation_t::union_:
  case operation_t::intersection:
    hash_ = ((children.left->get_hash() & 0xFFFFFFFF) * 2654435761) +
        children.right->get_hash() & 0xFFFFFFFF) * 2654435741);
    break;
  case operation_t::complement:
    hash_ = (children.left->get_hash() & 0xFFFFFFFF) * 2654435761;
    break;
  case operation_t::variable:
    hash_ = (variable_id_ & 0xFFFFFFFF) * 2654435761;
    break;
  default:
    assert(false && "Unrecognized.");
  }

  // Add the operation type to the hash.
  const auto op_code = static_cast<unsigned>(op_) + 1;
  hash_ += (op_code & 0xFFFFFFFF) * 2654435723;
}
```

Let τ_1 and τ_2 be two terms. Let h_1 be the precalculated hash of τ_1 and h_2 be the precalculated hash of τ_2. The equality check procedure follows the following steps:

- if $h_1 \neq h_2$, then the terms are not equal
- if $h_1 = h_2$, then recursively compare the children.

4.4.4 Conversion from AST

The formula building from an AST is straightforward. Recursive depth first iteration over the AST. For each AST node a coresponding formula/term node is constructed. The implementation could be found in *term.cpp* and *formula.cpp* in the repository.

5 Tableau Implementation

The Tableau process is a decision procedure. Recursively breaks down a given formula into basic components. Based on that a decision can be concluded. The recursive step breaks down a formula part into one or two subformulas. Continuously appling the recursive step produces a binary tree. The nodes are the formulas and the links represent the recursive step.

All formulas in a branch are considered to be in conjunction. Contradiction may arise in a same branch if there exists a formula and its negation.

The main principle of the tableau is to break complex formula into smaller ones until complementary pairs of atomic formulas are produced.

Definition: Signed formulas set

The signed formulas set consists only of signed formulas. The letter X is usually used for its representation.

5.1 Definition: Tableau Step

The *Tableau Step* takes as input a formula and a signed formulas set and produces as output one or two new formulas, depending on the operation. The signed formulas set consists of the broken down formulas by previous tableau steps. The output of the tableau step depends on the rule applied to the formula.

5.1.1 Rules

Only *negation conjunction* and *disjunction* operations will be handled. The implication and equivalence are converted.

Negation

$$\frac{\mathbb{T}(\neg\varphi), X}{\mathbb{F}(\varphi), X} \qquad \frac{\mathbb{F}(\neg\varphi), X}{\mathbb{T}(\varphi), X}$$

Conjunction

$$\frac{\mathbb{T}(\varphi \wedge \psi), X}{\mathbb{T}\varphi, \mathbb{T}\psi, X} \qquad \frac{\mathbb{F}(\varphi \wedge \psi), X}{\mathbb{F}\varphi, X \quad \mathbb{F}\psi, X}$$

Disjunction

$$\frac{\mathbb{T}(\varphi \vee \psi), X}{\mathbb{T}\varphi, X \quad \mathbb{T}\psi, X} \qquad \frac{\mathbb{F}(\varphi \vee \psi), X}{\mathbb{F}\varphi, \mathbb{F}\psi, X}$$

For our case the functionality of the tableau process shall be extended. If the branch is not closed there are additional calculations needed in order to verify that there is no contradiction. Namely to verify that there is no contradiction on Term level. This means that there exists a satisfiable model. This verification can be done in different manners. Depending on the algorithm type. The best way to think about it is that the tableau process returns a not-closed branch and if there is a model for the set of atomic formuas in this branch, then the formula is satisfiable, otherwise the tableau process proceeds with the next not-closed branch. If such branch does not exist then the formula is not satisfiable.

5.2 Implementation

The program implementation of the tableau method follows the standard tableau process. First interesting design decision is to keep all true signed formulas in one data set, and all false signed formulas in another data set. This enables fast searches whether a formula has been signed as true or false.

Definition: Signed Formula Collection

Let X be a set of formulas, then X is called signed formula collection if and only if all formulas in X are signed as true or all formulas are signed as false.

This collection is implemented with *std::unordered_set* (hashset), which stores the formulas by pointers to their root nodes. The hashing uses the node's precalculated hash. The comparing is via the node's *operator*==. That way different pointers to subformulas with same structure will be threated as identic.

The average complexity for search, insert and erase in this collection is O(1). There is no formula coping. So, almost no memory overhead for keeping the formulas in the set.

There are 8 signed formula collections:

- formulas_T - contains only non-atomic formulas signed as true
- formulas_F - contains only non-atomic formulas signed as false,
  For example, if $\neg\varphi$ is encountered as an output of the tableau step, then only φ is inserted into the formula_F
- contacts_T - contains only atomic contact formulas signed as true
- contacts_F - contains only atomic contact formulas signed as false
- zero_terms_T - contains only formulas of type $\varphi \leq \psi$ signed as true
- zero_terms_F - contains only formulas of type $\varphi \leq \psi$ signed as false
- measured_less_eq_T - contains only formulas of type $\varphi \leq_m \psi$ signed as true

- measured_less_eq_F - contains only formulas of type $\varphi \leq_m \psi$ signed as false

These collctions are unordered sets of points to the formulas/terms.

```
types.h
using formulas_t = std::unordered_set<const formula*, formula_ptr_hasher, formula_ptr_comparat
using terms_t = std::unordered_set<const term*, term_ptr_hasher, term_ptr_comparator>;
```

```
tableau.h

    formulas_t formulas_T_;
    formulas_t formulas_F_;
    formulas_t contacts_T_;
    formulas_t contacts_F_;
    terms_t zero_terms_T_;
    terms_t zero_terms_F_;
    formulas_t measured_less_eq_T_;
    formulas_t measured_less_eq_F_;
```

Definition: Formula Contradiction

Let φ be a signed formula, then φ is causing a contradiction if any of the following is true:

- φ is a non-atomic signed as true and φ belongs to $formulas\_F$
- φ is a non-atomic signed as false and φ belongs to $formulas\_T$
- φ is a contact formula signed as true and φ belongs to $contacts\_F$
- φ is a contact formula signed as false and φ belongs to $contacts\_T$
- φ is a zero terms formula signed as true and φ belongs to $zero\_terms\_F$
- φ is a zero terms formula signed as false and φ belongs to $zero\_terms\_T$
- φ is a measured less formula signed as true and φ belongs to $measured\_less\_eq\_F$
- φ is a measured less formula signed as false and φ belongs to $measured\_less\_eq\_T$

Invariant

At any time, all formulas in all eight signed formula collections do not contradict.

A contradiction may occure if a formula is split and some of the resulting components causes a contradiction.

Example

Let's assume that $contacts\_T = \{C(a,b)\}$ and let's have a look at the following formula $\mathbb{T}(T \wedge \neg C(a,b))$.
By the rules of decomposition, namely the ($\wedge$) rule produces $\mathbb{T}T$ and $\mathbb{T}\neg C(a,b)$.

Then the $\mathbb{T}\neg C(a,b)$ will be decomposed to $\mathbb{F}C(a,b)$ by the ($\neg$) rule, which causes a contradiction since $C(a,b)$ is already present in $contacts\_T$ formulas.

Tableau Algorithm

Given a formula φ, the following algorithm determines the atomic formulas in all branches of the tableau process.

As a first step if the formula φ is the constant F, then false is returned directly, otherwise the whole formula φ is inserted in $formulas\_T$.

Remarks

- true boolean value is used to represent the formula constant T
- false boolean value is used to represent the formula constant F
- Contact atomic formula is commutative, meaning that: $C(a,b) \iff C(b,a)$

Few lemmas which will provide a much more efficient contradiction finding in the tableau process.

Lemma: A

Let x be a term. Suppose that the atomic formula $x = 0$ has already been signed as true. Then marking the following formulas as true will lead to contradiction:

- C(x,y)
- C(y,x)

for any term y.

Lemma: A-inverse

Let x, y and z be terms. Suppose that the atomic formulas C(x,y) or C(z, x) has already been signed as true, then marking the formula x = 0 as true will lead to contradiction.

Time Complexity A and A-inverse

The algorithmic complexity to check whether a new formula leads to contradiction by Lemma A and Lemma A-inverse is done effectively. Namely in constant time with the usage of one new collection $contact\_T\_terms\_$. It keeps the terms of the true contacts. Namely the contacts in the collection $contacts\_T$. This means that for each $\mathbb{T}(C(x,y))$, the terms x and y are in the mentioned collection of true terms. The $contact\_T\_terms\_$ is a multiset and keeps track of all added terms, meaning that if the term x is added twice and then removed only once there will still be an entry of that x in the $contact\_T\_terms\_$ collection. Removing a true contact appears when moving up the tableau tree, i.e. switching to another branch.

To check if a new formula leads to contradiction by Lemma A or Lemma A-inverse the following method is used:

```
auto has_broken_contact_rule(const formula* f) const -> bool;
```

5.2.1 Handy methods

Searching

Search for formula signed as true

```
auto find_in_T(const formula* f) const -> bool
```

Checks existence of formula φ in any positive collection depending on the type of φ. Namely if φ is of type:

- $C(x, y)$: returns whether $\varphi \in contacts\_T$
- $x = 0$: returns whether $\varphi \in zero\_terms\_T$
- $x \leq_m y$: returns whether $\varphi \in measured\_less\_eq\_T$
- non-atomic formula: returns whether $\varphi \in formulas\_T$

Search for formula signed as false

```
auto find_in_F(const formula* f) const -> bool
```

Checks existence of formula φ in any negative collection depending on the type of φ. Namely if φ is of type:

- $C(x, y)$: returns whether $\varphi \in contacts\_F$
- $x = 0$: returns whether $\varphi \in zero\_terms\_F$
- $x \leq_m y$: returns whether $\varphi \in measured\_less\_eq\_F$
- non-atomic formula: returns whether $\varphi \in formulas\_F$

Adding

Mark formula as true

```
void add_formula_to_T(const formula* f)
```

Adds the formula φ as true in in the respective positive collection. Namely if φ is of type:

- C(x, y): φ is added to $contacts\_T$, and the terms x and y are added to the $contact\_T\_terms\_$
- $x = 0$: x is added in $zero\_terms\_T$
- $x \leq_m y$: φ is added to $measured\_less\_eq\_T$
- non-atomic formula: φ is added to $formulas\_T$

Mark formula as false

```
void add_formula_to_F(const formula* f)
```

Adds the formula φ as false in the respective negative collection. Namely if φ is of type:

- C(x, y): φ is added to $contacts\_F$
- $x = 0$: x is added in $zero\_terms\_F$
- $x \leq_m y$: φ is added to $measured\_less\_eq\_F$
- non-atomic formula: φ is added to $formulas\_F$

Removing

Remove formula signed as true

```
void remove_formula_from_T(const formula* f)
```

Removes the formula φ from the respective positive collection. Namely if φ is of type:

- C(x, y): φ is removed from $contacts\_T$, and the terms x and y are removed from the $contact\_T\_terms\_$.
- $x = 0$: x is removed from $zero\_terms\_T$
- $x \leq_m y$: φ is removed from $measured\_less\_eq\_T$
- non-atomic formula: φ is removed from $formulas\_T$

Remove formula signed as false

```
void remove_formula_from_F(const formula* f)
```

Removes the formula φ from the respective negative collection. Namely if φ is of type:

- C(x, y): φ is removed from $contacts\_F$
- $x = 0$: x is removed from $zero\_terms\_F$
- $x \leq_m y$: φ is removed from $measured\_less\_eq\_F$
- non-atomic formula: φ is removed from $formulas\_F$

Tableau Satisfiable Step Implementation

The Tableau satisfiable step is the whole tableau algorithm.

```
auto tableau::satisfiable_step() -> bool
{
    // The bottom of the recursive algorithm is when we have
    // only atomic formulas(which does not contradicts).
    // Then we can run algorithms for model construction.
    if(formulas_T_.empty() && formulas_F_.empty())
    {
        // This is the method which tries
        // to construct satisfiable model.
        return has_satisfiable_model();
    }

    if(!formulas_T_.empty())
    {
        // Choosing some formula to handle in this step.
        // If this branch does not produce a valid satisfiable path,
        // then this formula will be returned to formulas_T_.
        auto f = *formulas_T_.begin();

        const auto op = f->get_operation_type();
        if(op == op_t::negation)
        {
            // T(~X) -> F(X)
            auto X = f->get_left_child_formula();
            if(X->is_constant())
            {
                // F(T) is not satisfiable
                if(X->is_constant_true())
                {
                    return false;
                }
                // F(F) is satisfiable, continue with the rest.
                return satisfiable_step();
            }

            if(find_in_T(X))
            {
                // Contradiction, we want to satisfy F(X)
                // but we already have to satisfy T(X).
                return false;
            }

            if(find_in_F(X)) // Skip adding F(X) multiple times.
            {
                return satisfiable_step();
            }

            add_formula_to_F(X);
            auto res = satisfiable_step();
            // Revert it on the way back.
            remove_formula_from_F(X);
            return res;
        }
```

```
        if(op == op_t::conjunction)
        {
            // T(X & Y) -> T(X) & T(Y)
            T_conjunction_child X(*this, f->get_left_child_formula());
            T_conjunction_child Y(*this, f->get_right_child_formula());

            // Checks if X breaks the contact rule
            // or brings a contradiction
            if(!X.validate())
            {
                return false;
            }
            X.add_to_T(); // Adds X to T collection

            if(!Y.validate())
            {
                X.remove_from_T();
                return false;
            }
            Y.add_to_T();

            auto res = satisfiable_step();
            X.remove_from_T();
            Y.remove_from_T();

            return res;
        }

        assert(op == op_t::disjunction);
        // T(X v Y) -> T(X) v T(Y)
        auto X = f->get_left_child_formula();
        auto Y = f->get_right_child_formula();
        trace() << "Will split to two subtrees: "
                << *X << " and " << *Y;

        // T(T) is satisfiable and we can skip the other branch
        if(X->is_constant_true() || Y->is_constant_true())
        {
            trace() << "One of the childs is constant true";
            return satisfiable_step();
        }
```

```
        auto process_T_disj_child = [&](const formula* child) {
            if(child->is_constant_false() || // T(F) is not satisfiable
               find_in_F(child) || has_broken_contact_rule(child))
            {
                return false;
            }

            if(find_in_T(child)) // skip adding it multiple times
            {
                return satisfiable_step();
            }

            add_formula_to_T(child);
            const auto res = satisfiable_step();
            remove_formula_from_T(child);
            return res;
        };

        trace() << "Start of the left subtree: " << *X << " of " << *f;
        if(process_T_disj_child(X))
        {
            // There was no contradiction in the left path,
            // so there is no need to continue with the right path.
            return true;
        }

        trace() << "Start of the right subtree: " << *Y << " of " << *f;
        return process_T_disj_child(Y);
    }

    // Almost analogous but taking a formula from Fs

    // Choosing some formula to handle in this step.
    // If this branch does not produce a valid satisfiable path,
    // then this formula will be returned to formulas_F_
    auto f = *formulas_F_.begin();

    const auto op = f->get_operation_type();
    if(op == op_t::negation)
    {
        // F(~X) -> T(X)
        auto X = f->get_left_child_formula();
        if(X->is_constant())
        {
            // T(F) is not satisfiable
            if(X->is_constant_false())
            {
                return false;
            }
            // T(T) is satisfiable, continue with the rest
            return satisfiable_step();
        }
```

```
        if(find_in_F(X))
        {
            // Contradiction, we want to satisfy T(X)
            // but we already have to satisfy F(X).
            return false;
        }
        // We will add T(X) where X might be Contact or =0 term,
        // so we need to verify that we will not break the contact rule.
        if(has_broken_contact_rule(X))
        {
            return false;
        }

        if(find_in_T(X)) // skip adding it multiple times
        {
            return satisfiable_step();
        }

        add_formula_to_T(X);
        auto res = satisfiable_step();
        remove_formula_from_T(X);
        return res;
    }

    if(op == op_t::disjunction)
    {
        // F(X v Y) -> F(X) & F(Y)
        F_disjunction_child X(*this, f->get_left_child_formula());
        F_disjunction_child Y(*this, f->get_right_child_formula());

        // Checks that X does not bring a contradiction
        if(!X.validate())
        {
            return false;
        }
        X.add_to_F();

        if(!Y.validate())
        {
            X.remove_from_F();
            return false;
        }
        Y.add_to_F();

        auto res = satisfiable_step();

        X.remove_from_F();
        Y.remove_from_F();

        return res;
    }
```

```
    assert(op == op_t::conjunction);
    // F(X & Y) -> F(X) v F(Y)
    auto X = f->get_left_child_formula();
    auto Y = f->get_right_child_formula();

    trace() << "Will split to two subtrees: " << *X << " and " << *Y;

    // F(F) is satisfiable and we can skip the other branch
    if(X->is_constant_false() || Y->is_constant_false())
    {
        trace() << "One of the childs is constant false";
        return satisfiable_step();
    }

    auto process_F_conj_child = [&](const formula* child) {
        if(child->is_constant_true() || // F(T) is not satisfiable
           find_in_T(child))
        {
            return false;
        }
        if(find_in_F(child)) // skip adding it multiple times
        {
            return satisfiable_step();
        }

        add_formula_to_F(child);
        const auto res = satisfiable_step();
        remove_formula_from_F(child);
        return res;
    };

    trace() << "Start of the left subtree: " << *X << " of " << *f;
    if(process_F_conj_child(X))
    {
        // There was no contradiction in left path,
        // so there is no need to continue with the right path.
        return true;
    }

    trace() << "Start of the right subtree: " << *Y << " of " << *f;
    return process_F_conj_child(Y);
}
```

6 Model Implementation

Tableau branch output

As stated above the ouput of a branch in the tableau process is a set of atomic formulas. These atomic formulas are grouped in six sets:

- Contacts ($contacts\_T$)
- Non Contacts ($contacts\_F$)
- Equal to Zero Terms ($zero\_terms\_T$)
- Not Equal to Zero Terms ($zero\_terms\_F$)
- Meassured Equal to Zero Terms ($measured\_less\_eq\_T$)
- Meassured Not Equal to Zero Terms ($measured\_less\_eq\_F$)

All atomic formulas in the branch should be satisfied. So, they are in a conjunction. Can be represented with the following formula:

$$\bigwedge_i C(a_i, b_i) \wedge \bigwedge_j \neg C(e_j, f_j) \wedge$$
$$\bigwedge_k d_k = 0 \wedge \bigwedge_l g_l \neq 0 \wedge$$
$$\bigwedge_s <=_m (H_s, O_s) \wedge \bigwedge_u \neg(<=_m (Q_u, R_u))$$

Model output

The model bulding algorithm should produce a set of modal points. The contacts between them and to define the valuation for each boolean variable.

6.1 Modal point representation

The modal points are variable evaluations. The variables are converted to identificators from 0 to N - 1, where N is the number of different boolean variables. The variable evaluation is a sequence of N 1s and 0s. Thus, all different evaluations are 2^N. It is implemented via the *boost::dynamic_bitset*. Which is an optimized vector of N booleans. The memory for N elements is roughfly N bits. The element at position X is the evaluation for the variable with identificator X.

There might be variables in the formula which are not used in the branch conjunction. The evaluations for those variables are not needed. So, the variable evaluations will be only over the **used variables**. Let the used variables count is K. Then, all different modal points will be 2^K.

It is crucial to have an iterative algorithm for generating all modal points. The modal point representation is similar to the binary numbers. Therefore, the plus one binary operation is simulated over the bitset. It allows a generation of the next modal point. It is convenient for the model construction.

The following is the implementation of the variable evaluation:

```
variables_evaluations_block.h/cpp

using variables_mask_t = boost::dynamic_bitset<>;
using variables_evaluations_t = boost::dynamic_bitset<>;
using set_variables_ids_t = std::vector<size_t>;

class variables_evaluations_block {
public:
    variables_evaluations_block(const variables_mask_t& variables);

    auto get_variables() const -> variables_mask_t;
    auto get_evaluations() -> variables_evaluations_t&;
    auto get_evaluations() const -> const variables_evaluations_t&;

    auto get_set_variables_ids() const -> const set_variables_ids_t&;
    auto generate_next_evaluation() -> bool;
    void reset_evaluations();

private:
    void init();

    variables_mask_t variables_;
    variables_evaluations_t evaluations_;

    // Caching the set variables.
    // For generating the next evaluations in order to make it
    // O(|set varaibles|) instead of O(|all variables in the mask|)
    set_variables_ids_t set_variables_ids_;
};
```

```
...
auto variables_evaluations_block::generate_next_evaluation() -> bool
{
    if((variables_ & evaluations_) == variables_)
    {
        // If the evaluation for the variables is only 1s
        // then we cannot generate a new one,
        // i.e. we have already generated all of them.
        return false;
    }

    /*
     * Will generate the evaluations in the following order:
     * 0...00, 0...01, 0...10, ... , 11...10, 11...11.
     * This is very similar to the increment(+1) operation of integer
     * numbers in their binary representation.
     * For the binary number an algorithm could be the following:
     * Iterate all bits starting from the least significant.
     *     - bit(i) == 1 => bit(i) = 0
     *     - bit(i) == 0 => bit(i) = 1 & stop
     * In our case it is similar, we want to make the increment
     * operation only on the set bits in the variables_ mask.
     * set_variables_ids_ has the ids of the set bits
     * in the variables mask in reverce order.
     */
    for(const auto id : set_variables_ids_)
    {
        if(!evaluations_[id])
        {
            evaluations_.set(id);
            break;
        }
        else
        {
            evaluations_.set(id, false);
        }
    }

    return true;
}
```

```
model.h
using points_t = std::vector<variables_evaluations_block>;
points_t points_;
```

6.2 Contacts representation

The contact relations are impleneted via a standard adjacency matrix. The elements of the matrix indicate whether pairs of points are in contact or not. Their values are 0 or 1. Thus, the optimized *boost::dynamic_bitset* is used again.

```
using model_points_set_t = boost::dynamic_bitset<>;
using contacts_t = std::vector<model_points_set_t>;
contacts_t contact_relations_;
```

6.3 Valuation representation

The valuation v_n requires to define it for each boolean variable. It's implemeted via a NxM bit matrix. N is the number of boolean variables and M is the number of modal points. The matrix element at position (i, j) indicates whether the valuation for the variable with id i contains the modal point j.

```
using model_points_set_t = boost::dynamic_bitset<>;
using variable_id_to_points_t = std::vector<model_points_set_t>;

// A vector of bitsets representing the value of v(variable_id).
variable_id_to_points_t variable_evaluations_;
```

6.4 Handy methods

Contact matrix filling

The algorithm for building a model creates a pair of points for each contact in the branch conjunction. Therefore these points should be in contact. In addition to that each modal point is in contact with itself (reflexivity).

imodel.h/cpp

```
// Useful for models which have their first 2*@number_of_contacts points
// in contact (point 2k is in contact with point (2k+1))
// Inserts 1s in the contact relations matrix between points 2k and 2k+1
// (for each k in range [0, @number_of_contacts))
// Inserts 1s in the contact relations matrix between
// each point and itself (reflexivity).
void imodel::create_contact_relations_first_2k_in_contact(
        size_t number_of_points,
        size_t number_of_contacts)
{
    contact_relations_.clear();
    // Fill NxN matrix with 0s.
    contact_relations_.resize(number_of_points,
                              model_points_set_t(number_of_points));
    for(size_t k = 0; k < number_of_contacts; ++k)
    {
        const auto a = 2 * k;
        const auto b = a + 1;
        contact_relations_[a].set(b); // Sets the b-th bit to 1.
        contact_relations_[b].set(a);
    }

    // Add also the reflexivity.
    for(size_t i = 0; i < number_of_points; ++i)
    {
        contact_relations_[i].set(i);
    }
}
```

Variable evaluation filling

Fills the *variable_evaluations_* matrix based on the current modal points.

imodel.h/cpp

```
void model::calculate_the_model_evaluation_of_each_variable()
{
    const auto points_size = points_.size();
    variable_evaluations_.clear();
    // Initialize each variable evaluation as the empty set.
    variable_evaluations_.resize(
        mgr_->get_variables().size(),
        model_points_set_t(points_size));

    // Calculate the valuation of each variable,
    // i.e. each variable_id
    // v(Pi) = { point | point_evaluation[Pi] == 1 },
    // i.e. the evaluation of variable with id Pi is 1
    // (the bit at position Pi is 1)
    for(size_t point = 0; point < points_size; ++point)
    {
        const auto& point_evaluation = points_[point].get_evaluations();

        // Iterate only set bits(1s)
        auto Pi = point_evaluation.find_first();
        while(Pi != variables_evaluations_t::npos)
        {
            // Adds the point to the v(Pi) set.
            variable_evaluations_[Pi].set(point);
            Pi = point_evaluation.find_next(Pi);
        }
    }
}
```

Evaluating a term

The implementation of the boolean valuation **??** is in the *term* class. The details are in the *term.cpp* file. The boolean valuation assings a constant true or false to the term for some variable evaluation. This variable evaluation assings a constant true or false to each boolean variable in the term.

```
term.h

class term {
...
struct evaluation_result
{
    enum class result_type : char
    {
        none,
        constant_true,
        constant_false,
    };

    auto is_constant_true() const -> bool;
    auto is_constant_false() const -> bool;

    result_type type{result_type::none};
        ....
};
...
};

// Ignore the second argument for subterm creation.
// It is a support for a partial variable evaluation block
// which does not evaluate all boolean variables in the term.
// Then it will evaluate all known variables and reduces the constants.
// Returns it as a subterm.
// It is not used because it was needed
// for an old model building algorithm.
auto term::evaluate(
        const variables_evaluations_block& evaluation_block,
        bool skip_subterm_creation = true) const -> evaluation_result;
```

Zero terms satisfiability

Checks whether a modal point(variable evaluation) does not conflict with the zero terms. The point should not be part of any zero term evaluation. So, the point should evaluate all zero terms to constant false.

```
utils.h/cpp

/// Returns true if the evaluation evaluates all zero terms to false.
auto are_zero_terms_T_satisfied(
        const terms_t& zero_terms_T,
        const variables_evaluations_block& evaluation) -> bool
{
    // The evaluation should evaluate all zero terms to constant false.
    // That way it will not participate in any of their evaluations.
    for(const auto& z : zero_terms_T)
    {
        if(!z->evaluate(evaluation).is_constant_false())
        {
            return false;
        }
    }
    return true;
}
```

Non-contacts satisfiability

Checks whether a modal point (or a pair of points) does not conflict with the non-contacts. It is splitted to two components. Based on the reflexivity and connectivity rules.

For the reflexivity it is sufficient to verify that the point is not part of the both non-contact terms evaluations. So, the point should not evaluate both terms to constant true.

For the connectivity it is sufficient to verify that the pair of points does not participate in the non-contact terms evaluations. So, the points should not evaluate the terms to constant true.

```
utils.h/cpp
auto is_contacts_F_reflexive_rule_satisfied(
        const formulas_t& contacts_F,
        const variables_evaluations_block& evaluation) -> bool
{
    for(const auto& c : contacts_F)
    {
        // The evaluation should not be parth of both
        // non-contact term's evaluations.
        const auto left_t = c->get_left_child_term();
        const auto right_t = c->get_right_child_term();
        if(left_t->evaluate(evaluation).is_constant_true() &&
           right_t->evaluate(evaluation).is_constant_true())
        {
            return false;
        }
    }
    return true;
}

auto is_contacts_F_connectivity_rule_satisfied(
        const formulas_t& contacts_F,
        const variables_evaluations_block& eval_a,
        const variables_evaluations_block& eval_b) -> bool
{
    for(const auto& c : contacts_F)
    {
        // In order the eval_a and eval_b to not conflict with a
        // non-contact they should not participate in the non-contact
        // term's evaluations. In other words, both evaluations
        // should not evaluate both terms to true.
        const auto l = c->get_left_child_term();
        const auto r = c->get_right_child_term();
        if((l->evaluate(eval_a).is_constant_true() &&
            r->evaluate(eval_b).is_constant_true()) ||
           (l->evaluate(eval_b).is_constant_true() &&
            r->evaluate(eval_a).is_constant_true()))
        {
            // The reflexivity case is not taken into account here.
            return false;
        }
    }
    return true;
}
```

Modal points constructors

Construction modal points for non-zero terms

Creates a modal point for each non-zero term in the branch conjunction. The point should not conflict with any zero term or non-contact.

```
model.h/cpp

auto model::construct_non_zero_model_points(
        const terms_t& zero_terms_F, const formulas_t& contacts_F,
        const terms_t& zero_terms_T) -> bool
{
    for(const auto& z : zero_terms_F)
    {
        // It will be overriten if succeed.
        variables_evaluations_block eval(variables_mask_t(0));
        if(!create_point_evaluation(z, eval, contacts_F, zero_terms_T))
        {
            return false;
        }
        points_.push_back(std::move(eval));
    }

    return true;
}

auto model::create_point_evaluation(
        const term* t, variables_evaluations_block& out_evaluation,
        const formulas_t& contacts_F,
        const terms_t& zero_terms_T) const -> bool
{
    out_evaluation = variables_evaluations_block(used_variables_);

    return does_point_evaluation_satisfies_basic_rules(
                t, out_evaluation, contacts_F, zero_terms_T) ||
            generate_next_point_evaluation(
                t, out_evaluation, contacts_F, zero_terms_T);
}

auto model::does_point_evaluation_satisfies_basic_rules(
        const term* t,
        const variables_evaluations_block& evaluation,
        const formulas_t& contacts_F,
        const terms_t& zero_terms_T) const -> bool
{
    return t->evaluate(evaluation).is_constant_true() &&
           are_zero_terms_T_satisfied(zero_terms_T, evaluation) &&
           is_contacts_F_reflexive_rule_satisfied(
               contacts_F, evaluation);
}
```

```
auto model::are_zero_terms_T_satisfied(
        const terms_t& zero_terms_T,
        const variables_evaluations_block& evaluation) const -> bool
{
    for(const auto& z : zero_terms_T)
    {
        if(!z->evaluate(evaluation).is_constant_false())
        {
            return false;
        }
    }
    return true;
}

auto model::generate_next_point_evaluation(
        const term* t, variables_evaluations_block& out_evaluation,
        const formulas_t& contacts_F,
        const terms_t& zero_terms_T) const -> bool
{

    while(out_evaluation.generate_next_evaluation())
    {
        if(does_point_evaluation_satisfies_basic_rules(
              t, out_evaluation, contacts_F, zero_terms_T))
        {
            return true;
        }
    }
    return false;
}
```

Construction modal points for contacts

Creates a pair of modal points for each contact in the branch conjunction.

```
model.h/cpp

auto model::construct_contact_model_points(
        const formulas_t& contacts_T, const formulas_t& contacts_F,
        const terms_t& zero_terms_T) -> bool
{
    for(const auto& c : contacts_T)
    {
        if(!construct_contact_points(c, contacts_F, zero_terms_T))
        {
            return false;
        }
    }

    return true;
}
```

```
auto model::construct_contact_points(
        const formula* c, const formulas_t& contacts_F,
        const terms_t& zero_terms_T) -> bool
{
    const auto left = c->get_left_child_term();
    const auto right = c->get_right_child_term();

    // It will be overridden if succeed.
    variables_evaluations_block left_eval(variables_mask_t(0));
    if(!create_point_evaluation(
            left, left_eval, contacts_F, zero_terms_T))
    {
        return false;
    }

    do
    {
        variables_evaluations_block right_eval(variables_mask_t(0));
        if(!create_point_evaluation(
                right, right_eval, contacts_F, zero_terms_T))
        {
            return false;
        }

        do
        {
            if(is_contacts_F_connectivity_rule_satisfied(
                    contacts_F, left_eval, right_eval))
            {
                points_.push_back(std::move(left_eval));
                points_.push_back(std::move(right_eval));
                return true;
            }
        } while(generate_next_point_evaluation(
                    right, right_eval, contacts_F, zero_terms_T));
    } while(generate_next_point_evaluation(
                left, left_eval, contacts_F, zero_terms_T));

    return false;
}
```

6.5 Building algorithm

The building algorithm is simple. Creates a pair of suitable modal points for each contact. Creates a suitable modal point for each non-zero term. Lastly, updates the boolean variable valuation and connectivity matrix.

```
model.h/cpp

auto model::create(
        const formulas_t& contacts_T, const formulas_t& contacts_F,
        const terms_t& zero_terms_T, const terms_t& zero_terms_F,
        const variables_mask_t& used_variables,
        const formula_mgr* mgr) -> bool
{
    ...
    if(!construct_contact_model_points(
            contacts_T, contacts_F, zero_terms_T) ||
       !construct_non_zero_model_points(
           zero_terms_F, contacts_F, zero_terms_T))
    {
        return false;

    }

    if(points_.empty() &&
       !construct_point(contacts_F, zero_terms_T))
    {
        return false;
    }

    calculate_the_model_evaluation_of_each_variable();
    create_contact_relations_first_2k_in_contact(
        points_.size(), contacts_T.size());
    return true;
}
```

7 Connected Contact Logic Implementation

The connected model reuses the core part of the model. It has the same modal points contacts and valuation representation. The model output is also the same. Uses some of the described Handy methods in the Model implementation section.

7.1 Handy methods

Construction of all valid modal points

The first step of the algorithm for building a connected model is to create all valid modal points. A valid modal point is a point that does not conflict with any zero term or non-contact.

```
connected_model.h/cpp

void connected_model::construct_all_valid_unique_points(
        const formulas_t& contacts_F,
        const terms_t& zero_terms_T)
{
    variables_evaluations_block evaluation(used_variables_);

    do
    {
        if(are_zero_terms_T_satisfied(zero_terms_T, evaluation) &&
           is_contacts_F_rule_satisfied_only_reflexivity(
               contacts_F, evaluation))
        {
            points_.push_back(evaluation);
        }
    } while(evaluation.generate_next_evaluation());
}
```

Evaluating a term with a valuation

The term evaluation with the *variable_evaluation_block* returns a constant true or false. It tells whether the variable evaluation participates in the term's evaluation.

Another way of evaluating a term is via the whole valuation. It returns a set of modal points which participate in its evaluation. It is implemented in the *term* class.

```
term.h/cpp

auto term::evaluate(
        const variable_id_to_points_t& variable_evaluations,
        const size_t points_count) const -> model_points_set_t
{
    switch(op_)
    {
        case operation_t::constant_true:
            return ~model_points_set_t(points_count);
        case operation_t::constant_false:
            return model_points_set_t(points_count);
        case operation_t::union_:
            return childs_.left->evaluate(variable_evaluations,
                                          points_count) |
                   childs_.right->evaluate(variable_evaluations,
                                           points_count);
        case operation_t::intersection:
            return childs_.left->evaluate(variable_evaluations,
                                          points_count) &
                   childs_.right->evaluate(variable_evaluations,
                                           points_count);
        case operation_t::complement:
            return ~childs_.left->evaluate(variable_evaluations,
                                           points_count);
        case operation_t::variable:
            // Returns the evaluation for the variable.
            return variable_evaluations[variable_id_];
        default:
            assert(false && "Unrecognized.");
            return model_points_set_t(points_count);
    }
}
```

Contacts satisfcation

Checks whether the contacts are satisfied. It is sufficient to verify that there is a pair of points in the contact terms evaluations which are in contact.

```
connected_model.h/cpp

auto connected_model::is_contacts_T_rule_satisfied(
        const formulas_t& contacts_T) const -> bool
{
    // C(a,b)
    for(const auto& c : contacts_T)
    {
        if(!is_contact_satisfied(c))
        {
            return false;
        }
    }
    return true;
}

auto connected_model::is_contact_satisfied(
        const formula* c) const -> bool
{
    const auto left_t = c->get_left_child_term();
    const auto right_t = c->get_right_child_term();
    const auto v_a = left_t->evaluate(variable_evaluations_,
                                      points_.size());
    const auto v_b = right_t->evaluate(variable_evaluations_,
                                       points_.size());

    auto point_from_v_a = v_a.find_first();
    while (point_from_v_a != model_points_set_t::npos)
    {
        const auto& contacts_of_point_from_v_a =
            contact_relations_[point_from_v_a];
        if ((contacts_of_point_from_v_a & v_b).any())
        {
            return true;
        }

        point_from_v_a = v_a.find_next(point_from_v_a);
    }
    return false;
}
```

Contacts existence satisfiability

Checks whether the contacts have at least one point in their term's evaluations. It's a faster verification than the whole contact satisfication.

```
connected_model.h/cpp

auto connected_model::is_contacts_T_existence_rule_satisfied(
        const formulas_t& contacts_T) const -> bool
{
    for(const auto& c : contacts_T)
    {
        const auto a = c->get_left_child_term();
        const auto b = c->get_right_child_term();

        if(a->evaluate(variable_evaluations_, points_.size()).none() ||
           b->evaluate(variable_evaluations_, points_.size()).none())
        {
            return false;
        }
    }
    return true;
}
```

Non-zero term satisfiability

Checks whether the non-zero terms are satisfied. It is sufficient to verify that there is a at least one modal point in each non-zero term's evaluation.

```
connected_model.h/cpp

auto connected_model::is_zero_terms_F_rule_satisfied(
        const terms_t& zero_terms_F) const -> bool
{
    for(const auto& z : zero_terms_F)
    {
        if(z->evaluate(variable_evaluations_, points_.size()).none())
        {
            return false;
        }
    }
    return true;
}
```

Contact matrix building

The first step of the algorithm for building a connected model is to create all valid modal points. A valid modal point is a point which does not conflict with any zero term or non-contact.

connected_model.h/cpp

```
/*
* Builds N x N bit matrix with contact relations between all points,
* which later reduces in order to satisfy all ~C atomic formulas.
* Returns true if the produced contact relations satisfy all
* contact atomic formulas.
*/
auto connected_model::build_contact_relations_matrix(
    const formulas_t& contacts_T, const formulas_t& contacts_F) -> bool
{
    const auto point_size = points_.size();
    contact_relations_.clear();
    // Fill NxN matrix with 1s.
    contact_relations_.resize(point_size,
                              ~model_points_set_t(point_size));

    // Removes the connection between every point in v_l
    // and all points from v_r.
    auto connection_remover = [&](auto& v_l, auto& v_r)
    {
        const auto neg_v_r = ~v_r;

        auto point_from_v_l = v_l.find_first();
        while (point_from_v_l != model_points_set_t::npos)
        {
            auto& contacts_of_point_from_v_l =
                contact_relations_[point_from_v_l];
            // Remove all points in @contacts_of_point_from_v_l
            // which are also in @v_r.
            contacts_of_point_from_v_l &= neg_v_r;

            point_from_v_l = v_l.find_next(point_from_v_l);
        }
    };
```

```
    // ~C(a,b)
    for(const auto& c : contacts_F)
    {
        const auto left_t = c->get_left_child_term();
        const auto right_t = c->get_right_child_term();
        const auto v_a = left_t->evaluate(variable_evaluations_,
                                          points_.size());
        const auto v_b = right_t()->evaluate(variable_evaluations_,
                                             points_.size());
                // There are no common points,
                // because we built them that way.
        assert((v_a & v_b).none());

        connection_remover(v_a, v_b);
        connection_remover(v_b, v_a);
    }

    return is_contacts_T_rule_satisfied(contacts_T);
}
```

Connected components

Let us look at the model as a graph. The modal points are the vertices. The contacts between them are the edges. This method returns a vector of connected components in the graph. Each connected component is described by the points in it. The algorithm is a standard BFS from each not visited node in the graph.

```
connected_model.h/cpp

auto connected_model::get_connected_components()
        const -> std::vector<model_points_set_t>
{
    if(points_.empty())
    {
        return {};
    }

    std::vector<model_points_set_t> connected_components;
    model_points_set_t not_visited_points(points_.size());
    // Bitset of 1s for the not visited points, inverted
    // because we have fast finding of 1s in the bitset.
    not_visited_points.set();

    size_t root_point_id = not_visited_points.find_first();
    while(root_point_id != model_points_set_t::npos)
    {
        auto connected_component =
            get_connected_component(root_point_id, not_visited_points);
        connected_components.push_back(std::move(connected_component));

        root_point_id = not_visited_points.find_next(root_point_id);
    }
    return connected_components;
}
```

```
auto connected_model::get_connected_component(
        size_t root_point_id,
        model_points_set_t& not_visited_points)
        const -> model_points_set_t
{
    assert(not_visited_points.test(root_point_id));

    model_points_set_t connected_component(points_.size());

    // A simple traversing
    std::queue<size_t> q;
    q.push(root_point_id);

    while(!q.empty())
    {
        const auto point_id = q.front();
        q.pop();
        // While the point waits in the queue some other point
        // could also push it.
        if(!not_visited_points.test(point_id))
        {
            continue;
        }

        const auto& point_connections = contact_relations_[point_id];
        auto connected_point_id = point_connections.find_first();
        while(connected_point_id != model_points_set_t::npos)
        {
            if(not_visited_points.test(connected_point_id))
            {
                connected_component.set(point_id);
                not_visited_points.reset(point_id);
                q.push(connected_point_id);
            }
            connected_point_id =
                point_connections.find_next(connected_point_id);
        }
    }

    return connected_component;
}
```

Reduce variable evaluations to a subset of points

Leaves only a selected subset of points in the variable evaluations.

```
connected_model.h/cpp
void connected_model::reduce_variable_evaluations_to_subset_of_points(
        const model_points_set_t& points_subset)
{
    for(auto& evaluation : variable_evaluations_)
    {
        evaluation &= points_subset;
    }
}
```

Reduce model to a subset of points

Leaves only a selected subset of points in the model. Updates the variable evaluations and the contact relations between them.

```
connected_model.h/cpp

void connected_model::reduce_model_to_subset_of_points(
        const model_points_set_t& points_subset)
{
    // Construct the reduced points collection.
    points_t reduced_points;
    const auto reduced_points_size = points_subset.count();
    reduced_points.reserve(reduced_points_size);
    // Let K is the number of points in @points_subset.
    // All collections should have size K. That require a mapping
    // between old points indexes and the reduced.
    std::unordered_map<size_t, size_t> point_id_old_to_new;
    point_id_old_to_new.reserve(reduced_points_size);
    auto point = points_subset.find_first();
    while (point != model_points_set_t::npos)
    {
        point_id_old_to_new[point] = reduced_points.size();

        reduced_points.push_back(points_[point]);
        point = points_subset.find_next(point);
    }

    // Converts the @points_subset of old points to a subset of
    // reduced points. The old points have a (potential) bigger
    // container size than the reduced points.
    auto map_subset_of_points_to_reduced_points =
        [&](const model_points_set_t& points_subset)->model_points_set_t
    {
        // All 0s.
        model_points_set_t mapped_subset(reduced_points_size);

        auto point = points_subset.find_first();
        while (point != model_points_set_t::npos)
        {
            mapped_subset.set(point_id_old_to_new[point]);
            point = points_subset.find_next(point);
        }
        return mapped_subset;
    };
```

```
    // KxK matrix of 0s.
    contacts_t reduced_contact_relations(
        reduced_points_size,
        model_points_set_t(reduced_points_size));

    point = points_subset.find_first();
    while (point != model_points_set_t::npos)
    {
        auto& point_contacts = contact_relations_[point];
        point_contacts &= points_subset; // Remove extra contacts.

        const auto reduced_point_id = point_id_old_to_new[point];
        reduced_contact_relations[reduced_point_id] =
            map_subset_of_points_to_reduced_points(point_contacts);
        point = points_subset.find_next(point);
    }

    // Update the models data.
    points_ = std::move(reduced_points);
    contact_relations_ = std::move(reduced_contact_relations);
    calculate_the_model_evaluation_of_each_variable();
}
```

7.2 Building algorithm

The building algorithm used the described methods. Follows the 3.2 connected model building algorithm.

```
connected_model.h/cpp

auto connected_model::create(
        const formulas_t& contacts_T, const formulas_t& contacts_F,
        const terms_t& zero_terms_T, const terms_t& zero_terms_F,
        const formulas_t&, const formulas_t&,
        const variables_mask_t& used_variables,
        const formula_mgr* mgr) -> bool
{
    construct_all_valid_unique_points(contacts_F, zero_terms_T);

    if(points_.empty())
    {
        trace() << "Unable to create even one model point which "
        "does not break the =0 or the reflexivity of ~C atomics.";
        return false;
    }

    calculate_the_model_evaluation_of_each_variable();

    if(!is_zero_terms_F_rule_satisfied(zero_terms_F) ||
       !is_contacts_T_existence_rule_satisfied(contacts_T))
    {
        trace() << "Unable to create model points even to satisfy "
        "the existence of points in the model evaluation "
        "of != 0 and contact terms.";
        return false;
    }

    if(!build_contact_relations_matrix(contacts_T, contacts_F))
    {
        trace() << "Unable to create contact relations which "
        "satisfies ~C and C atomic formulas.";
        return false;
    }

    // Now, we have in some sense the biggest model (w.r.t number
    // of unique points and maximal contact relations between them).
    trace() << "Will try to find a connected component of points "
    "(which itself is a satisfiable model) in the following "
    "satisfiable model:\n" << *this;
```

```
    const auto original_variable_evaluations = variable_evaluations_;
    const auto connected_components = get_connected_components();
    for(const auto& connected_component : connected_components)
    {
        // Restrict the variable evaluations to only those points
        // which are in the @connected_component.
        reduce_variable_evaluations_to_subset_of_points(
            connected_component);

        if(is_zero_terms_F_rule_satisfied(zero_terms_F) &&
           is_contacts_T_rule_satisfied(contacts_T))
        {
            // Good. The connected component is also a valid model.
            // Remove all other points(outside the connected component)
            // because we do not need them.
            reduce_model_to_subset_of_points(connected_component);
            trace() << "Found a connected component which is also "
            "a satisfiable model for the formula:\n" << *this;
            return true;
        }

        // Rollback the variable_evaluations.
        variable_evaluations_ = original_variable_evaluations;
    }

    trace() << "Unable to find such connected component of points.";
    return false;
}
```

References

[1] Handbook of Tableau Methods M. D'Agostino, D. M. Gabbay, R. H¨ahnle and J. Posegga, eds.

[2] Modal Logics for Region-based Theories of Space Philippe Balbiani, Thinko Tinchev, Dimiter Vakarelov

[3] Connected Space from Wikipedia

[4] Flex & Bison overview by aquamentus

[5] Flex & Bison by John Levine.

[6] C++ Rest SDK by Microsoft

[7] Flex tokenizer

[8] Bison parser

[9] Abstract syntax tree (AST) from Wikipedia

[10] Visitor pattern from Wikipedia